L2/19-360

 Technical Reports

Proposed Update

Unicode® Standard Annex #15

UNICODE NORMALIZATION FORMS

Version	Unicode 13.0.0
Editors	Ken Whistler (ken@unicode.org)
Date	2019-05-15
This Version	http://www.unicode.org/reports/tr15/tr15-49.html
Previous Version	http://www.unicode.org/reports/tr15/tr15-48.html
Latest Version	http://www.unicode.org/reports/tr15/
Latest Proposed Update	http://www.unicode.org/reports/tr15/proposed.html
Revision	49

Summary

This annex describes normalization forms for Unicode text. When implementations keep strings in a normalized form, they can be assured that equivalent strings have a unique binary representation. This annex also provides examples, additional specifications regarding normalization of Unicode text, and information about conformance testing for Unicode normalization forms.

Status

This is a draft document which may be updated, replaced, or superseded by other documents at any time. Publication does not imply endorsement by the Unicode Consortium. This is not a stable document; it is inappropriate to cite this document as other than a work in progress.

A Unicode Standard Annex (UAX) forms an integral part of the Unicode Standard, but is published online as a separate document. The Unicode Standard may require conformance to normative content in a Unicode Standard Annex, if so specified in the Conformance chapter of that version of the Unicode Standard. The version number of a UAX document corresponds to the version of the Unicode Standard of which it forms a part.

Please submit corrigenda and other comments with the online reporting form [Feedback]. Related information that is useful in understanding this annex is found in Unicode Standard Annex #41, "Common References for Unicode Standard Annexes." For the latest version of the Unicode Standard, see [Unicode]. For a list of current Unicode Technical Reports, see [Reports]. For more information about versions of the Unicode Standard, see [Versions]. For any errata which may apply to this annex, see [Errata].

Contents

---

1 Introduction

This annex provides subsidiary information about Unicode normalization. It describes canonical and compatibility equivalence and the four normalization forms, providing examples, and elaborates on the formal specification of Unicode normalization, with further explanations and implementation notes.

This document also provides the formal specification of the Stream-Safe Text Format and of the Normalization Process for Stabilized Strings.

For the formal specification of the Unicode Normalization Algorithm, see *Section 3.11, Normalization Forms* in [Unicode].

For a general introduction to the topic of equivalent sequences for Unicode strings and the need for normalization, see *Section 2.12, Equivalent Sequences and Normalization* in [Unicode].

1.1 Canonical and Compatibility Equivalence

The Unicode Standard defines two formal types of equivalence between characters: *canonical equivalence* and *compatibility equivalence*. Canonical equivalence is a fundamental equivalency between characters or sequences of characters which represent the same abstract character, and which when correctly displayed should always have the same visual appearance and behavior. *Figure 1* illustrates this type of equivalence with examples of several subtypes.

Figure 1. Examples of Canonical Equivalence

Subtype	Examples
Combining sequence	Ç ⟷ C + ◌̧
Ordering of combining marks	q + ◌̇ + ◌̣ ⟷ q + ◌̣ + ◌̇
Hangul & conjoining jamo	가 ⟷ ㄱ + ㅏ
Singleton equivalence	Ω ⟷ Ω

Compatibility equivalence is a weaker type of equivalence between characters or sequences of characters which represent the same abstract character (or sequence of abstract characters), but which may have distinct visual appearances or behaviors. The visual appearances of the compatibility equivalent forms typically constitute a subset of the expected range of visual appearances of the character (or sequence of characters) they are equivalent to. However, these variant forms may represent a visual distinction that is significant in some textual contexts, but not in others. As a result, greater care is required to determine when use of a compatibility equivalent is appropriate. If the visual distinction is stylistic, then markup or styling could be used to represent the formatting information. However, some characters with compatibility decompositions are used in mathematical notation to represent a distinction of a semantic nature; replacing the use of distinct character codes by formatting in such contexts may cause problems. *Figure 2* provides examples of compatibility equivalence.

Figure 2. Examples of Compatibility Equivalence

Subtype	Examples
Font variants	ℌ → H

	ℍ → H	
Linebreaking differences	[NBSP] → [SPACE]	
Positional variant forms	‎ع‎ → ‎ع‎	
	‎ع‎ → ‎ع‎	
	‎ع‎ → ‎ع‎	
	‎ع‎ → ‎ع‎	
Circled variants	① → 1	
Width variants	カ → カ	
Rotated variants	⏜ → {	
	⏝ → }	
Superscripts/subscripts	i⁹ → i9	
	i₉ → i9	
Squared characters	㌀ → アパート	
Fractions	¼ → 1/4	
Other	dž → dž	

Both canonical and compatibility equivalences are explained in more detail in *Chapter 2, General Structure*, and *Chapter 3, Conformance,* in [Unicode].

1.2 Normalization Forms

Unicode Normalization Forms are formally defined normalizations of Unicode strings which make it possible to determine whether any two Unicode strings are equivalent to each other. Depending on the particular Unicode Normalization Form, that equivalence can either be a canonical equivalence or a compatibility equivalence.

Essentially, the Unicode Normalization Algorithm puts all combining marks in a specified order, and uses rules for decomposition and composition to transform each string into one of the Unicode Normalization Forms. A binary comparison of the transformed strings will then determine equivalence.

The four Unicode Normalization Forms are summarized in *Table 1.*

Table 1. Normalization Forms

Form	Description
Normalization Form D (NFD)	Canonical Decomposition
Normalization Form C (NFC)	Canonical Decomposition, followed by Canonical Composition
Normalization Form KD (NFKD)	Compatibility Decomposition
Normalization Form KC (NFKC)	Compatibility Decomposition, followed by Canonical Composition

There are two forms of normalization that convert to composite characters: *Normalization Form C* and *Normalization Form KC*. The difference between these depends on whether the resulting text is to be a *canonical* equivalent to the original unnormalized text or a *compatibility* equivalent to the original unnormalized text. (In *NFKC* and *NFKD,* a *K* is used to stand for *compatibility* to avoid confusion with the *C* standing for *composition*.) Both types of normalization can be useful in different circumstances.

Figures 3 through *6* illustrate different ways in which source text can be normalized. In the first three figures, the NFKD form is always the same as the NFD form, and the NFKC form is always the same as the NFC form, so for simplicity those columns are omitted. For consistency, all of these examples use Latin characters, although similar examples are found in other scripts.

Figure 3. Singletons

Certain characters are known as singletons. They never remain in the text after normalization. Examples include the *angstrom* and *ohm* symbols, which map to their normal letter counterparts *a-with-ring* and *omega*, respectively.

Figure 4. Canonical Composites

Many characters are known as canonical composites, or precomposed characters. In the D forms, they are decomposed; in the C forms, they are *usually* precomposed. (For exceptions, see *Section 5, Composition Exclusion Table*.)

Normalization provides a unique order for combining marks, with a uniform order for all D and C forms. Even when there is no precomposed character, as with the "q" with accents in *Figure 5*, the ordering may be modified by normalization.

Figure 5. Multiple Combining Marks

The example of the letter "d" with accents shows a situation where a precomposed character plus another accent changes in NF(K)C to a *different* precomposed character plus a different accent.

Figure 6. Compatibility Composites

Source		NFD	NFC	NFKD	NFKC
fi FB01	:	fi FB01	fi FB01	f i 0066 0069	f i 0066 0069
2^5 0032 2075	:	2 5 0032 2075	2 5 0032 2075	2 5 0032 0035	2 5 0032 0035
ẛ 1E9B 0323	:	ſ ○ ○ 017F 0323 0307	ẛ ○ 1E9B 0323	s ○ ○ 0073 0323 0307	ṩ 1E69

In the NFKC and NFKD forms, many formatting distinctions are removed, as shown in *Figure 6*. The "fi" ligature changes into its components "f" and "i", the superscript formatting is removed from the "5", and the long "s" is changed into a normal "s".

Normalization Form KC does *not* attempt to map character sequences to compatibility composites. For example, a compatibility composition of "office" does *not* produce "o\uFB03ce", even though "\uFB03" is a character that is the compatibility equivalent of the sequence of three characters "ffi". In other words, the composition phase of NFC and NFKC are the same—only their decomposition phase differs, with NFKC applying compatibility decompositions.

Normalization Form C uses canonical composite characters where possible, and maintains the distinction between characters that are compatibility equivalents. Typical strings of composite accented Unicode characters are already in Normalization Form C. Implementations of Unicode that restrict themselves to a repertoire containing no combining marks are already typically using Normalization Form C. (Implementations need to be aware of versioning issues—see *Section 3, Versioning and Stability*.)

The *W3C Character Model for the World Wide Web 1.0: Normalization* [CharNorm] and other W3C Specifications (such as XML 1.0 5th Edition) recommend using Normalization Form C for all content, because this form avoids potential interoperability problems arising from the use of canonically equivalent, yet different, character sequences in document formats on the Web. See the *W3C Character Model for the Word Wide Web: String Matching and Searching* [CharMatch] for more background.

Normalization Form KC additionally folds the differences between compatibility-equivalent characters that are inappropriately distinguished in many circumstances. For example, the halfwidth and fullwidth *katakana* characters will normalize to the same strings, as will Roman numerals and their letter equivalents. More complete examples are provided in *Section 6, Examples and Charts*.

Normalization Forms KC and KD must *not* be blindly applied to arbitrary text. Because they erase many formatting distinctions, they will prevent round-trip conversion to and from many legacy character sets, and unless supplanted by formatting markup, they may remove distinctions that are important to the semantics of the text. It is best to think of these Normalization Forms as being like uppercase or lowercase mappings: useful in certain contexts for identifying core meanings, but also performing modifications to the text that may not always be appropriate. They can be applied more freely to domains with restricted

character sets. (See Unicode Standard Annex #31, "Unicode Identifier and Pattern Syntax" [UAX31] for examples.)

To summarize the treatment of compatibility composites that were in the source text:

- Both NFD and NFC maintain compatibility composites.
- Neither NFKD nor NFKC maintains compatibility composites.
- None of the forms *generate* compatibility composites that were not in the source text.

For a list of all characters that may change in any of the Normalization Forms (aside from reordering), see the Normalization Charts [Charts15].

1.3 Description of the Normalization Process

This section provides a short summary of how the Unicode Normalization Algorithm works.

To transform a Unicode string into a given Unicode Normalization Form, the first step is to fully decompose the string. The decomposition process makes use of the Decomposition_Mapping property values defined in UnicodeData.txt. There are also special rules to fully decompose Hangul syllables. Full decomposition involves recursive application of the Decomposition_Mapping values, because in some cases a complex composite character may have a Decomposition_Mapping into a sequence of characters, one of which may also have its own non-trivial Decomposition_Mapping value.

The type of full decomposition chosen depends on which Unicode Normalization Form is involved. For NFC or NFD, one does a full *canonical* decomposition, which makes use of only canonical Decomposition_Mapping values. For NFKC or NFKD, one does a full *compatibility* decomposition, which makes use of canonical *and* compatibility Decomposition_Mapping values.

Once a string has been fully decomposed, any sequences of combining marks that it contains are put into a well-defined order. This rearrangement of combining marks is done according to a subpart of the Unicode Normalization Algorithm known as the Canonical Ordering Algorithm. That algorithm sorts sequences of combining marks based on the value of their Canonical_Combining_Class (ccc) property, whose values are also defined in UnicodeData.txt. Most characters (including all non-combining marks) have a Canonical_Combining_Class value of zero, and are unaffected by the Canonical Ordering Algorithm. Such characters are referred to by a special term, *starter*. Only the subset of combining marks which have non-zero Canonical_Combining_Class property values are subject to potential reordering by the Canonical Ordering Algorithm. Those characters are called *non-starters*.

At this point, if one is transforming a Unicode string to NFD or NFKD, the process is complete. However, one additional step is needed to transform the string to NFC or NFKC: recomposition. The fully decomposed and canonically ordered string is processed by another subpart of the Unicode Normalization Algorithm known as the Canonical Composition Algorithm. That process logically starts at the front of the string and systematically checks it for pairs of characters which meet certain criteria and for which there is a canonically equivalent composite character in the standard. Each appropriate pair of characters which meet the criteria is replaced by the composite character, until the string contains no further such pairs. This transforms the fully decomposed string into its most fully *composed* but still canonically equivalent sequence.

Figure 7 shows a sample of how the composition process works. The gray cubes represent starters, and the white cubes represent non-starters. In the first step, the string is fully decomposed and canonically reordered. This is represented by the downwards arrows. In the

second step, each character is checked against the last non-starter and starter, and combined if all the appropriate conditions are met. This is represented by the curved arrows pointing to the starters. Note that in each case, all of the successive white boxes (non-starters) are examined *plus* one additional gray box (starter). *Examples* are provided in *Section 6, Examples and Charts*.

Figure 7. Composition Process
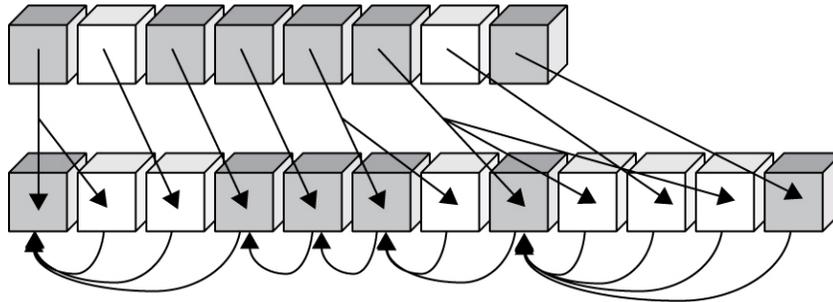
Taken step-by-step, the Unicode Normalization Algorithm is fairly complex. However, it is designed in such a way that it enables very efficient, highly-optimized implementations. For example, checking whether a Unicode string is in NFC is a very quick process, and since much text is already in NFC, an implementation that normalizes strings to NFC mostly consists of quick verification checks, with only very occasional modifications of any pieces which are not already in NFC. See *Section 9, Detecting Normalization Forms*.

> *Note:* Text exclusively containing ASCII characters (U+0000..U+007F) is left unaffected by all of the Normalization Forms. This is particularly important for programming languages. (See Unicode Standard Annex #31, "Unicode Identifier and Pattern Syntax" [UAX31].) Text exclusively containing Latin-1 characters (U+0000..U+00FF) is left unaffected by NFC. This is effectively the same as saying that all Latin-1 text is *already* normalized to NFC.

The complete formal specification of the Unicode Normalization Algorithm and of the Unicode Normalization Forms can be found in *Section 3.11, Normalization Forms* in [Unicode]. See that section for all of the formal definitions and for the details of the exact formulation of each step in the algorithm.

1.4 Concatenation of Normalized Strings

In using normalization functions, it is important to realize that *none* of the Normalization Forms are closed under string concatenation. That is, even if two strings X and Y are normalized, their string concatenation X+Y is *not* guaranteed to be normalized. This even happens in NFD, because accents are canonically ordered, and may rearrange around the point where the strings are joined. Consider the string concatenation examples shown in *Table 2*.

Table 2. String Concatenation

Form	String1	String2	Concatenation	Correct Normalization
NFD	a ◌̂	◌̣	a ◌̂ ◌̣	a ◌̣ ◌̂
NFC	a	◌̂	a ◌̂	â
NFC	ㄱ	ㅏ ㄴ	ㄱ ㅏ ㄴ	각

However, it is possible to produce an optimized function that concatenates two normalized strings and *does* guarantee that the result is normalized. Internally, it only needs to normalize characters around the boundary of where the original strings were joined, within stable code points. For more information, see *Section 9.1, Stable Code Points.*

In contrast to their behavior under string concatenation, all of the Normalization Forms *are* closed under substringing. For example, given a substring of a normalized string X, from offsets 5 to 10, the resulting string will still be normalized.

2 Notation

Table 3 lists examples of the notational conventions used in this annex.

Table 3. Notational Conventions

Example Notation	Description
"...\uXXXX..."	The Unicode character U+XXXX embedded within a string
k_i, a_m, and k_f	Conjoining jamo types (initial, medial, final) represented by subscripts
NFx	Any Unicode Normalization Form: NFD, NFKD, NFC, or NFKC
toNFx(s)	A function that produces the the normalized form of a string s according to the definition of Unicode Normalization Form X
isNFx(s)	A binary property of a string s, whereby: isNFx(s) is true if and only if toNFX(s) is identical to s. See also *Section 9, Detecting Normalization Forms.*
X ≈ Y	X is canonically equivalent to Y
X[*i, j*]	The substring of X that includes all code units after offset *i* and before offset *j*; for example, if X is "abc", then X[1,2] is "b"

Additional conventions used in this annex:

1. A sequence of characters may be represented by using plus signs between the character names or by using string notation.
2. An *offset into a Unicode string* is a number from 0 to *n*, where *n* is the length of the string and indicates a position that is logically between Unicode code units (or at the very front or end in the case of 0 or *n*, respectively).
3. Unicode names may be shortened, as shown in *Table 4.*

Table 4. Character Abbreviation

Abbreviation	Full Unicode Name
E-grave	LATIN CAPITAL LETTER E WITH GRAVE

ka	KATAKANA LETTER KA
hw_ka	HALFWIDTH KATAKANA LETTER KA
ten	COMBINING KATAKANA–HIRAGANA VOICED SOUND MARK
hw_ten	HALFWIDTH KATAKANA VOICED SOUND MARK

3 Versioning and Stability

It is crucial that Normalization Forms remain stable over time. That is, if a string that does not have any unassigned characters is normalized under one version of Unicode, it must remain normalized under all future versions of Unicode. This is the backward compatibility requirement. To meet this requirement, a fixed version for the composition process is specified, called the *composition version.* The composition version is defined to be **Version 3.1.0** of the Unicode Character Database. For more information, see

- Versions of the Unicode Standard [Versions]
- Unicode 3.1 [Unicode3.1]
- Unicode Character Database [UCD]

To see what difference the composition version makes, suppose that a future version of Unicode were to add the composite *Q-caron*. For an implementation that uses that future version of Unicode, strings in Normalization Form C or KC would continue to contain the sequence *Q + caron,* and *not* the new character *Q-caron*, because a canonical composition for *Q-caron* was not defined in the composition version. See *Section 5, Composition Exclusion Table*, for more information.

It would be possible to add more compositions in a future version of Unicode, as long as the backward compatibility requirement is met. It requires that for any new composition XY → Z, at most one of X or Y was defined in a previous version of Unicode. That is, Z must be a new character, and either X or Y must be a new character. However, the Unicode Consortium strongly discourages new compositions, even in such restricted cases.

In addition to fixing the composition version, future versions of Unicode must be restricted in terms of the kinds of changes that can be made to character properties. Because of this, the Unicode Consortium has a clear policy to guarantee the stability of Normalization Forms.

The Unicode Consortium has well-defined policies in place to govern changes that affect backward compatibility. According to the Unicode policy for Normalization Forms, applicable to Unicode 4.1 and all later versions, the results of normalizing a string on one version will always be the same as normalizing it on any other version, as long as the string contains only assigned characters according to both versions. For information on these stability policies, especially regarding normalization, see the Unicode Character Encoding Stability Policy [Policies].

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 5.0 program normalizes a string that contains new Unicode 5.1 characters. That string might not be normalized according to Unicode 5.1.

Prior to Unicode 4.1, the stability policy was not quite as strict. For more information, see *Section 11 Stability Prior to Unicode 4.1.*

4 Conformance

Starting with Unicode 5.2.0, conformance clauses UAX15-C1 and UAX15-C2 have been redirected to point to the formal specification of Unicode Normalization Forms in *Section 3.11, Normalization Forms* in [Unicode]. All of the local clauses have been retained in this annex, so that any external references to Unicode Standard Annex #15 and to particular conformance clauses for Unicode Normalization Forms will continue to be valid. Specific references to any definitions used by the Unicode Normalization Algorithm also remain valid.

UAX15-C1. *A process that produces Unicode text that purports to be in a Normalization Form shall do so in accordance with the specifications in Section 3.11, Normalization Forms in [Unicode].*

- See C13 in *Chapter 3, Conformance* in [Unicode]

UAX15-C2. *A process that tests Unicode text to determine whether it is in a Normalization Form shall do so in accordance with the specifications in Section 3.11, Normalization Forms in [Unicode]*

- See C14 in *Chapter 3, Conformance* in [Unicode]

UAX15-C3. *A process that purports to transform text into a Normalization Form must be able to produce the results of the conformance test specified in the NormalizationTest.txt data file [Test15].*

- See C15 in *Chapter 3, Conformance* in [Unicode]
- The NormalizationText.txt file consists of a series of fields. When Normalization Forms are applied to the different fields in the test file, the results shall be as specified in the header of that file.

UAX15-C4. *A process that purports to transform text according to the Stream-Safe Text Format must do so in accordance with the specifications in this annex.*

UAX15-C5. *A process that purports to transform text according to the Normalization Process for Stabilized Strings must do so in accordance with the specifications in this annex.*

The specifications for Normalization Forms are written in terms of a process for producing a decomposition or composition from an arbitrary Unicode string. This is a *logical* description— particular implementations can have more efficient mechanisms as long as they produce the same result. See C18 in *Chapter 3, Conformance* in [Unicode] and the notes following.

Implementations must be thoroughly tested for conformance to the normalization specification. Testing for a particular Normalization Form does not require directly applying the process of normalization, so long as the result of the test is equivalent to applying normalization and then testing for binary identity.

5 Composition Exclusion

The concept of *composition exclusion* is a key part of the Unicode Normalization Algorithm. For normalization forms NFC and NFKC, which normalize Unicode strings to **C**omposed forms, where possible, the basic process is first to fully decompose the string, and then to compose the string, except where blocked or excluded. (See D117, Canonical Composition Algorithm, in Section 3.11, Normalization Forms in [Unicode].) This section provides information about the types of characters which are excluded from composition during

application of the Unicode Normalization Algorithm, and describes the data files which provide the definitive lists of those characters.

Composition exclusion characters have an associated binary character property in the [UCD]: Composition_Exclusion. It is a notable characteristic of the Unicode Normalization Algorithm that no composition exclusion character can occur in any normalized form of Unicode text: NFD, NFC, NFKD, or NFKC.

5.1 Composition Exclusion Types

Four types of canonically decomposable characters are excluded from composition in the Canonical Composition Algorithm. These four types are described and exemplified here.

Script-specific Exclusions

The term *script-specific exclusion* refers to certain canonically decomposable characters for which the non-decomposed form is generally not the preferred form for that script whose decomposition includes one of a small set of combining marks for particular Indian scripts, for Tibetan, or for Hebrew.

The list of such characters cannot be computed from the decomposition mappings in the Unicode Character Database, and must instead be explicitly listed.

The character U+0958 (क़) DEVANAGARI LETTER QA is an example of a script-specific composition exclusion.

The list of script-specific composition exclusions constituted a one-time adjustment to the Unicode Normalization Algorithm, defined at the time of the composition version in 2001 and unchanged since that version. The list consists essentially of precomposed characters using a *nukta* diacritic in various Indic scripts, and precomposed Hebrew characters involving *dagesh* or other marks. can be divided into the following three general groups, all added to the Unicode Standard before Version 3.1:

- Most precomposed characters using a *nukta* diacritic in the Bangla/Bengali, Devanagari, Gurmukhi, or Odia/Oriya scripts, mostly consisting of additions to the core set of letters for those scripts.
- Precomposed characters using the Tibetan lenition mark, U+0F39 *tsa-phru*, plus two two-part vowels which should not be composed in Tibetan.
- A large collection of compatibility precomposed characters for Hebrew involving *dagesh* and/or other combining marks.

Although, in principle, the list of script-specific composition exclusions could be expanded to add newly encoded characters in future versions of the Unicode Standard, it is very unlikely to be extended for such characters, because the preferred form of sequences is normalization forms of sequences are now taken into account *before* new characters are encoded.

Post Composition Version Exclusions

The term *post composition version exclusion* refers to certain canonical decomposable characters which were added after the composition version, and which meet certain criteria for exclusion.

The list of such characters cannot be computed from the decomposition mappings in the Unicode Character Database, and must instead be explicitly listed.

A canonical decomposable character *must* be added to the list of post composition version exclusions when its decomposition mapping is defined to contain at least one character which was already encoded in an earlier version of the Unicode Standard. This criterion is designed to minimize the encoding of unnecessary precomposed forms, but it also precludes certain conditions which could break backwards compatibility for the Unicode Normalization Algorithm.

An example of such a post composition version exclusion is U+2ADC (⫝) FORKING. To date, that one character, encoded in Unicode 3.2, is the *only* character added to the list of composition exclusions based on the criterion of its decomposition mapping containing a prior-encoded character.

A canonical decomposable character *may* be added to the list of post composition version exclusions when its decomposition mapping is defined to contain only characters which are first encoded in same version of the Unicode Standard as the canonical decomposable character, itself.

An example of such a post composition version exclusion is U+1D15F (♩) MUSICAL SYMBOL QUARTER NOTE. To date, that character and a related set of musical note symbols, encoded in Unicode 3.1, are the *only* characters added to the list of composition exclusions based on the criterion of their decomposition mappings containing only characters encoded in the same version of the Unicode Standard. Note that, techically, the encoding of those particular musical symbols did not formally postdate the composition version, but that fact is now a historical oddity resulting from early uncertainty as to whether the composition version would be fixed at Unicode 3.0 or Unicode 3.1.

In principle, future canonical decomposable characters could be added to the list of post composition version exclusions, if the UTC determines that their preferred representation is a decomposed sequence. In practice, this situation has not actually occurred since the publication of Unicode 3.1, and is unlikely to occur in the future, given current practice for assigning decomposition mappings for newly encoded characters.

Singleton Exclusions

A singleton decomposition is defined as a canonical decomposition mapping from a character to different single character. (See D110 in Section 3.11, Normalization Forms in [Unicode].) Characters which have single decompositions are automatically excluded from composition in the Canonical Composition Algorithm.

The list of characters with singleton decompositions is directly derivable from the list of decomposition mappings in the Unicode Character Database. For information, that list is also provided in comment lines in CompositionExclusions.txt in the UCD.

An example of a singleton exclusion is U+2126 (Ω) OHM SIGN.

There are cases where two characters have the same canonical decomposition in the Unicode Character Database. *Table 5* shows an example.

Table 5. Same Canonical Decomposition

Character	Full Decomposition
212B (Å) ANGSTROM SIGN	0041 (A) LATIN CAPITAL LETTER A + 030A (°) COMBINING RING ABOVE
00C5 (Å) LATIN CAPITAL LETTER A	

WITH RING ABOVE	

In such a case, the practice is to assign a singleton decomposition for one character to the other. The full decomposition for both characters then is derived from the decomposition mapping for the second character. In this particular case U+112B ANGSTROM SIGN has a singleton decomposition to U+00C5 LATIN CAPITAL LETTER A WITH RING ABOVE. Instances of characters with such singleton decompositions occur in the Unicode Standard for compatibility with certain pre-existing character encoding standards.

Non-starter Decomposition Exclusions

A non-starter decomposition is defined as an expanding canonical decomposition which is not a starter decomposition. (See D110b and D111 in Section 3.11, Normalization Forms in [Unicode].) Characters which have non-starter decompositions are automatically excluded from composition in the Canonical Composition Algorithm.

The list of characters with non-starter decompositions is directly derivable from the list of decomposition mappings in the Unicode Character Database. For information, that list is also provided in comment lines in CompositionExclusions.txt in the UCD.

An example of a non-starter decomposition exclusion is U+0344 (ः) COMBINING GREEK DIALYTIKA TONOS.

5.2 Composition Exclusion Data Files

The list of composition exclusion characters (Composition_Exclusion = True) is available as a machine-readable data file, CompositionExclusions.txt [Exclusions] in the Unicode Character Database [UCD].

All four classes of composition exclusion characters are included in this file, although the singletons and non-starter decompositions are provided in comment lines, as they can be computed directly from the decomposition mappings in the Unicode Character Database.

A derived property containing the complete list of full composition exclusion characters (Full_Composition_Exclusion = True), is available separately in the Unicode Character Database [UCD] and is described in Unicode Standard Annex #44, "Unicode Character Database" [UAX44]. Implementations can avoid having to compute the singleton and non-starter decompositions from the Unicode Character Database by using the Full_Composition_Exclusion property instead.

6 Examples and Charts

This section provides some detailed examples of the results when each of the Normalization Forms is applied. The Normalization Charts [Charts15] provide charts of all the characters in Unicode that differ from at least one of their Normalization Forms (NFC, NFD, NFKC, NFKD).

Basic Examples

The basic examples in *Table 6* do not involve compatibility decompositions. Therefore, in each case Normalization Forms NFD and NFKD are identical, and Normalization Forms NFC and NFKC are also identical.

Table 6. Basic Examples

Original	NFD, NFKD	NFC, NFKC	Notes

a	D–dot_above	D + dot_above	D–dot_above	Both decomposed and precomposed canonical sequences produce the same result.
b	D + dot_above	D + dot_above	D–dot_above	
c	D–dot_below + dot_above	D + dot_below + dot_above	D–dot_below + dot_above	The *dot_above* cannot be combined with the D because the D has already combined with the intervening *dot_below*.
d	D–dot_above + dot_below	D + dot_below + dot_above	D–dot_below + dot_above	
e	D + dot_above + dot_below	D + dot_below + dot_above	D–dot_below + dot_above	
f	D + dot_above + horn + dot_below	D + horn + dot_below + dot_above	D–dot_below + horn + dot_above	There may be intervening combining marks, so long as the result of the combination is canonically equivalent.
g	E–macron–grave	E + macron + grave	E–macron–grave	Multiple combining characters are combined with the base character.
h	E–macron + grave	E + macron + grave	E–macron–grave	
i	E–grave + macron	E + grave + macron	E–grave + macron	Characters will *not* be combined if they would not be canonical equivalents because of their ordering.
j	angstrom_sign	A + ring	A–ring	Because Å (A–ring) is the preferred composite, it is the form produced for both characters.
k	A–ring	A + ring	A–ring	

Effect of Compatibility Decompositions

The examples in *Table 7* and *Table 8* illustrate the effect of compatibility decompositions. When text is normalized in forms NFD and NFC, as in *Table 7*, compatibility-equivalent strings do not result in the same strings. However, when the same strings are normalized in forms NFKD and NFKC, as shown in *Table 8*, they do result in the same strings. The tables also contain an entry showing that Hangul syllables are maintained under all Normalization Forms.

Table 7. NFD and NFC Applied to Compatibility-Equivalent Strings

Original	NFD	NFC	Notes

l	"Äffin"	"A\u0308ffin"	"Äffin"	The *ffi_ligature* (U+FB03) is *not* decomposed, because it has a compatibility mapping, not a canonical mapping. (See *Table 8*.)
m	"Ä\uFB03n"	"A\u0308\uFB03n"	"Ä\uFB03n"	
n	"Henry IV"	"Henry IV"	"Henry IV"	Similarly, the ROMAN NUMERAL IV (U+2163) is *not* decomposed.
o	"Henry \u2163"	"Henry \u2163"	"Henry \u2163"	
p	ga	ka + ten	ga	Different compatibility equivalents of a single Japanese character will *not* result in the same string in NFC.
q	ka + ten	ka + ten	ga	
r	hw_ka + hw_ten	hw_ka + hw_ten	hw_ka + hw_ten	
s	ka + hw_ten	ka + hw_ten	ka + hw_ten	
t	hw_ka + ten	hw_ka + ten	hw_ka + ten	
u	kaks	k_i + a_m + ks_f	kaks	Hangul syllables are maintained under normalization.

Table 8. NFKD and NFKC Applied to Compatibility-Equivalent Strings

	Original	NFKD	NFKC	Notes
l'	"Äffin"	"A\u0308ffin"	"Äffin"	The *ffi_ligature* (U+FB03) *is* decomposed in NFKC (where it is not in NFC).
m'	"Ä\uFB03n"	"A\u0308ffin"	"Äffin"	
n'	"Henry IV"	"Henry IV"	"Henry IV"	Similarly, the resulting strings here are identical in NFKC.
o'	"Henry \u2163"	"Henry IV"	"Henry IV"	
p'	ga	ka + ten	ga	Different compatibility equivalents of a single Japanese character *will* result in the same string in NFKC.
q'	ka + ten	ka + ten	ga	
r'	hw_ka + hw_ten	ka + ten	ga	
s'	ka + hw_ten	ka + ten	ga	
t'	hw_ka + ten	ka + ten	ga	

u'	kaks	$k_i + a_m + ks_f$	kaks	Hangul syllables are maintained under normalization.*

* In earlier versions of Unicode, jamo characters like ks_f had compatibility mappings to $k_f + s_f$. These mappings were removed in Unicode 2.1.9 to ensure that Hangul syllables would be maintained.

7 Design Goals

The following are the design goals for the specification of the Normalization Forms and are presented here for reference. The first goal is a fundamental conformance feature of the design.

Goal 1: Uniqueness

The first, and by far the most important, design goal for the Normalization Forms is uniqueness. Two equivalent strings will have *precisely* the same normalized form. More explicitly,

1. If two strings x and y are canonical equivalents, then
   toNFC(x) = toNFC(y)
   toNFD(x) = toNFD(y)
2. If two strings are compatibility equivalents, then
   toNFKC(x) = toNFKC(y)
   toNFKD(x) = toNFKD(y)
3. All of the transformations are idempotent: that is,
   toNFC(toNFC(x)) = toNFC(x)
   toNFD(toNFD(x)) = toNFD(x)
   toNFKC(toNFKC(x)) = toNFKC(x)
   toNFKD(toNFKD(x)) = toNFKD(x)

Goal 1.3 is a consequence of Goals 1.2 and 1.1, but is stated here for clarity.

Another consequence of the definitions is that any chain of normalizations is equivalent to a single normalization, which is:

1. a compatibility normalization, if *any* normalization is a compatibility normalization
2. a composition normalization, if the *final* normalization is a composition normalization

For example, the following table lists equivalent chains of two transformations:

toNFC(x)	toNFD(x)	toNFKC(x)	toNFKD(x)
=toNFC(toNFC(x))	=toNFD(toNFC(x))	=toNFC(toNFKC(x))	=toNFD(toNFKC(x))
=toNFC(toNFD(x))	=toNFD(toNFD(x))	=toNFC(toNFKD(x))	=toNFD(toNFKD(x))
		=toNFKC(toNFC(x))	=toNFKD(toNFC(x))
		=toNFKC(toNFD(x))	=toNFKD(toNFD(x))

| | | =toNFKC(toNFKC(x)) | =toNFKD(toNFKC(x)) |
| | | =toNFKC(toNFKD(x)) | =toNFKD(toNFKD(x)) |

Goal 2: Stability

The second major design goal for the Normalization Forms is stability of characters that are not involved in the composition or decomposition process.

1. If a string x contains a character with a compatibility decomposition, then toNFD(x) and toNFC(x) still contain that character.
2. As much as possible, if there are no combining characters in x, then toNFC(x) = x.
   - The only characters for which this is not true are those in the *Section 5, Composition Exclusion Table*.
3. Irrelevant combining marks should not affect the results of composition. See example **f** in *Section 6, Examples and Charts,* where the *horn* character does not affect the results of composition.

Goal 3: Efficiency

The third major design goal for the Normalization Forms is to allow efficient implementations.

1. It is possible to implement efficient code for producing the Normalization Forms. In particular, it should be possible to produce Normalization Form C very quickly from strings that are already in Normalization Form C or are in Normalization Form D.
2. Normalization Forms that compose do not have to produce the shortest possible results, because that can be computationally expensive.

8 Legacy Encodings

While the Normalization Forms are specified for Unicode text, they can also be extended to non-Unicode (legacy) character encodings. This is based on mapping the legacy character set strings to and from Unicode using definitions UAX15-D1 and UAX15-D2.

UAX15-D1. An *invertible transcoding* T for a legacy character set L is a one-to-one mapping from characters encoded in L to characters in Unicode with an associated mapping T^{-1} such that for any string S in L, $T^{-1}(T(S)) = S$.

Most legacy character sets have a single invertible transcoding in common use. In a few cases there may be multiple invertible transcodings. For example, Shift-JIS may have two different mappings used in different circumstances: one to preserve the '\' semantics of $5C_{16}$, and one to preserve the '¥' semantics.

The character indexes in the legacy character set string may be different from character indexes in the Unicode equivalent. For example, if a legacy string uses visual encoding for Hebrew, then its first character might be the last character in the Unicode string.

If transcoders are implemented for legacy character sets, it is recommended that the result be in Normalization Form C where possible. See Unicode Technical Standard #22, "Unicode Character Mapping Markup Language" [UTS22] for more information.

UAX15-D2. Given a string S encoded in L and an invertible transcoding T for L, the *Normalization Form X of S under T* is defined to be the result of mapping to Unicode,

normalizing to Unicode Normalization Form X, and mapping back to the legacy character encoding—for example, T^{-1}(toNFx(T(S))). Where there is a single invertible transcoding for that character set in common use, one can simply speak of the Normalization Form X of S.

Legacy character sets are classified into three categories based on their normalization behavior with accepted transcoders.

1. *Prenormalized.* Any string in the character set is already in Normalization Form X.
   - For example, ISO 8859-1 is prenormalized in NFC.
2. *Normalizable.* Although the set is not prenormalized, any string in the set *can* be normalized to Normalization Form X.
   - For example, ISO 2022 (with a mixture of ISO 5426 and ISO 8859-1) is normalizable.
3. *Unnormalizable.* Some strings in the character set cannot be normalized into Normalization Form X.
   - For example, ISO 5426 is unnormalizable in NFC under common transcoders, because it contains combining marks but not composites.

9 Detecting Normalization Forms

The Unicode Character Database supplies properties that allow implementations to quickly determine whether a string x is in a particular Normalization Form—for example, isNFC(x). This is, in general, many times faster than normalizing and then comparing.

For each Normalization Form, the properties provide three possible values for each Unicode code point, as shown in *Table 9*.

Table 9. Description of Quick_Check Values

Values	Abbr	Description
NO	N	The code point cannot occur in that Normalization Form.
YES	Y	The code point is a starter and can occur in the Normalization Form. In addition, for NFKC and NFC, the character may compose with a following character, but it *never* composes with a previous character.
MAYBE	M	The code point can occur, subject to canonical ordering, but with constraints. In particular, the text may not be in the specified Normalization Form depending on the context in which the character occurs.

Code that uses this property can do a *very* fast first pass over a string to determine the Normalization Form. The result is also either NO, YES, or MAYBE. For NO or YES, the answer is definite. In the MAYBE case, a more thorough check must be made, typically by putting a copy of the string into the Normalization Form and checking for equality with the original.

- Even the slow case can be optimized, with a function that does not perform a complete normalization of the entire string, but instead works incrementally, only normalizing a limited area around the MAYBE character. See *Section 9.1, Stable Code Points*.

This check is much faster than simply running the normalization algorithm, because it avoids any memory allocation and copying. The vast majority of strings will return a definitive YES or NO answer, leaving only a small percentage that require more work. The sample below is written in Java, although for accessibility it avoids the use of object-oriented techniques.

```java
public int quickCheck(String source) {
    short lastCanonicalClass = 0;
    int result = YES;
    for (int i = 0; i < source.length(); ++i) {
        int ch = source.codepointAt(i);
        if (Character.isSupplementaryCodePoint(ch)) ++i;
        short canonicalClass = getCanonicalClass(ch);
        if (lastCanonicalClass > canonicalClass && canonicalClass != 0) {
            return NO;          }
        int check = isAllowed(ch);
        if (check == NO) return NO;
        if (check == MAYBE) result = MAYBE;
        lastCanonicalClass = canonicalClass;
    }
    return result;
}

public static final int NO = 0, YES = 1, MAYBE = -1;
```

The `isAllowed()` call should access the data from Derived Normalization Properties file [NormProps] for the Normalization Form in question. (For more information, see Unicode Standard Annex #44, "Unicode Character Database" [UAX44].) For example, here is a segment of the data for NFC:

```
...
0338 ; NFC_QC; M # Mn COMBINING LONG SOLIDUS OVERLAY
...

F900..FA0D ; NFD_QC; N # Lo [270] CJK COMPATIBILITY IDEOGRAPH-F900..CJK COMPATIBILITY IDEOGRAPH-FA0D
...
```

These lines assign the value NFC_QC==MAYBE to the code point U+0338, and the value NFC_QC==NO to the code points in the range U+F900..U+FA0D. There are no MAYBE values for NFD and NFKD: the `quickCheck` function will always produce a definite result for these Normalization Forms. All characters that are not specifically mentioned in the file have the values YES.

The data for the implementation of the `isAllowed()` call can be accessed in memory with a hash table or a trie (see *Section 14, Implementation Notes*); the latter will be the fastest.

There is also a Unicode Consortium stability policy that canonical mappings are always limited in all versions of Unicode, so that no string when decomposed with NFC expands to more than 3× in length (measured in code units). This is true whether the text is in UTF-8, UTF-16, or UTF-32. This guarantee also allows for certain optimizations in processing, especially in determining buffer sizes. See also *Section 13, Stream-Safe Text Format*.

9.1 Stable Code Points

It is sometimes useful to distinguish the set of code points that are *stable* under a particular Normalization Form. They are the set of code points never affected by that particular normalization process. This property is very useful for skipping over text that does not need to be considered at all, either when normalizing or when testing normalization.

Formally, each stable code point CP fulfills *all* of the following conditions:

1. CP has canonical combining class 0.

2. CP is (as a single character) not changed by this Normalization Form.

In case of NFC or NFKC, each stable code point CP fulfills *all* of the following additional conditions:

3. CP can never compose with a previous character.
4. CP can never compose with a following character.
5. CP can never change if another character is added.

Example. In NFC, *a-breve* satisfies all but (5), but if one adds an *ogonek* it changes to *a-ogonek* plus *breve*. So *a-breve* is not stable in NFC. However, *a-ogonek* is stable in NFC, because it does satisfy (1–5).

Concatenation of normalized strings to produce a normalized result can be optimized using stable code points. An implementation can find the last stable code point L in the first string, and the first stable code point F in the second string. The implementation has to normalize only the range from (and including) L to the last code point before F. The result will then be normalized. This can be a very significant savings in performance when concatenating large strings.

Because characters with the property values Quick_Check=YES and Canonical_Combining_Class=0 satisfy conditions 1–3, the optimization can also be performed using the Quick_Check property. In this case, the implementation finds the last code point L with Quick_Check=YES and Canonical_Combining_Class=0 in the first string and the first code point F with Quick_Check=YES and Canonical_Combining_Class=0 in the second string. It then normalizes the range of code points starting from (and including) L to the code point just before F.

10 Respecting Canonical Equivalence

This section describes the relationship of normalization to respecting (or preserving) canonical equivalence. A process (or function) *respects* canonical equivalence when canonical-equivalent inputs always produce canonical-equivalent outputs. For a function that transforms one string into another, this may also be called *preserving* canonical equivalence. There are a number of important aspects to this concept:

1. The outputs are *not* required to be identical, only canonically equivalent.
2. *Not* all processes are required to respect canonical equivalence. For example:
   - A function that collects a set of the General_Category values present in a string will and should produce a different value for <*angstrom sign, semicolon*> than for <*A, combining ring above, greek question mark*>, even though they are canonically equivalent.
   - A function that does a binary comparison of strings will also find these two sequences different.
3. Higher-level processes that transform or compare strings, or that perform other higher-level functions, must respect canonical equivalence or problems will result.

The canonically equivalent inputs or outputs are not just limited to strings, but are also relevant to the *offsets* within strings, because those play a fundamental role in Unicode string processing.

Offset P into string X is canonically equivalent to offset Q into string Y if and only if both of the following conditions are true:

X[0, P] ≈ Y[0, Q], and

$$X[P, \text{len}(X)] \approx Y[Q, \text{len}(Y)]$$

This can be written as $P_X \approx Q_Y$. Note that whenever X and Y are canonically equivalent, it follows that $0_X \approx 0_Y$ and $\text{len}(X)_X \approx \text{len}(Y)_Y$.

Example 1. Given X = *<angstrom sign, semicolon>* and Y = *<A, combining ring above, greek question mark>*,

$0_X \approx 0_Y$

$1_X \approx 2_Y$

$2_X \approx 3_Y$

1_Y has no canonically equivalent offset in X

The following are examples of processes that involve canonically equivalent strings *and/or* offsets.

Example 2. When `isWordBreak(string, offset)` respects canonical equivalence, then

`isWordBreak(`*<A-ring, semicolon>*`, 1) = isWordBreak(`*<A, ring, semicolon>*`, 2)`

Example 3. When `nextWordBreak(string, offset)` respects canonical equivalence, then

`nextWordBreak(`*<A-ring, semicolon>*`, 0) = 1` if and only if `nextWordBreak(`*<A, ring, semicolon>*`, 0) = 2`

Respecting canonical equivalence is related to, but different from, preserving a canonical Normalization Form NFx (where NFx means either NFD or NFC). In a process that preserves a Normalization Form, whenever any input string is normalized according to that Normalization Form, then every output string is also normalized according to that form. A process that preserves a canonical Normalization Form respects canonical equivalence, but the reverse is not necessarily true.

In building a system that as a whole respects canonical equivalence, there are two basic strategies, with some variations on the second strategy.

A. Ensure that each system component respects canonical equivalence.
B. Ensure that each system component preserves NFx, and one of the following:
    1. Reject any non-NFx text on input to the whole system.
    2. Reject any non-NFx text on input to each component.
    3. Normalize to NFx all text on input to the whole system.
    4. Normalize to NFx all text on input to each component.
    5. All three of the following:
        a. Allow text to be marked as NFx when generated.
        b. Normalize any unmarked text on input to each component to NFx.
        c. Reject any marked text that is not NFx.

There are trade-offs for each of these strategies. The best choice or mixture of strategies will depend on the structure of the components and their interrelations, and how fine-grained or low-level those components are. One key piece of information is that it is much faster to check that text is NFx than it is to convert it. This is especially true in the case of NFC. So even where it says "normalize" above, a good technique is to first check if normalization is required, and perform the extra processing only if necessary.

- Strategy A is the most robust, but may be less efficient.
- Strategies B1 and B2 are the most efficient, but would reject some data, including that converted 1:1 from some legacy code pages.
- Strategy B3 does not have the problem of rejecting data. It can be more efficient than A: because each component is assured that all of its input is in a particular Normalization Form, it does not need to normalize, except internally. But it is less robust: any component that fails can "leak" unnormalized text into the rest of the system.
- Strategy B4 is more robust than B1 but less efficient, because there are multiple points where text needs to be checked.
- Strategy B5 can be a reasonable compromise; it is robust but allows for all text input.

11 Stability Prior to Unicode 4.1

For versions prior to Unicode 4.1 (that do not apply Corrigenda #2 through #5), slightly weaker stability policies are in effect. For information on these stability policies, especially regarding normalization, see the Unicode Character Encoding Stability Policy [Policies].

These policies still guaranteed, in particular, that:

Once a character is encoded, its canonical combining class and decomposition mapping will not be changed in a way that will destabilize normalization.

What this means is:

If a string contains only characters from a given version of the Unicode Standard (for example, Unicode 3.1.1), and it is put into a normalized form in accordance with that version of Unicode, then it will be in normalized form according to any future version of Unicode.

This guarantee has been in place for Unicode 3.1 and after. It has been necessary to correct the decompositions of a small number of characters since Unicode 3.1, as listed in the Normalization Corrections data file [Corrections], but such corrections are in accordance with the above principles: all text normalized on old systems will test as normalized in future systems. All text normalized in future systems will test as normalized on past systems. Prior to Unicode 4.1, what may change for those few characters, is that *unnormalized* text may normalize differently on past and future systems.

11.1 Stability of Normalized Forms

For all versions, even prior to Unicode 4.1, the following policy is followed:

A normalized string is guaranteed to be stable; that is, once normalized, a string is normalized according to all future versions of Unicode.

More precisely, if a string has been normalized according to a particular version of Unicode *and* contains only characters allocated in that version, it will qualify as normalized according to any future version of Unicode.

11.2 Stability of the Normalization Process

For all versions, even prior to Unicode 4.1, the *process* of producing a normalized string from an unnormalized string has the same results under each version of Unicode, except for certain edge cases addressed in the following corrigenda:

- Three corrigenda correct certain data mappings for a total of seven characters:

| Corrigendum #2, "U+FB1D Normalization" [Corrigendum2] |
| Corrigendum #3, "U+F951 Normalization" [Corrigendum3] |
| Corrigendum #4, "Five Unihan Canonical Mapping Errors" [Corrigendum4] |

- Corrigendum #5, "Normalization Idempotency" [Corrigendum5], fixed a problem in the description of the normalization process for some instances of particular sequences. *Such instances never occur in meaningful text.*

11.3 Guaranteeing Process Stability

The Unicode Standard provides a mechanism for those implementations that require not only normalized strings, *but also the normalization process*, to be absolutely stable between two versions even prior to Unicode 4.1 (including the edge cases mentioned in *Section 11.2, Stability of the Normalization Process*). This, of course, is true only where the repertoire of characters is limited to those characters present in the earlier version of Unicode.

To have the newer implementation produce the same results as the older version (for characters defined as of the older version):

1. Premap a maximum of seven (rare) characters according to whatever corrigenda came between the two versions (see [Errata]).
   - For example, for a Unicode 4.0 implementation to produce the same results as Unicode 3.2, the five characters mentioned in [Corrigendum4] are premapped to the *old* values given in version 4.0 of the UCD data file [Corrections].
2. Apply the later version of normalization.
   - Handle any code points that were not defined in the earlier version as if they were unassigned: such code points will not decompose or compose, and their Canonical_Combining_Class value will be zero.
   - The Derived_Age property in the Unicode Character Database can be used to determine whether a code point is assigned for any particular version of the standard.
3. If the earlier version is before Unicode 4.1 and the later version is 4.1 or later and if the normalization is to forms NFC or NFKC, perform the following steps:
   - Reorder the sequences listed in *Table 10* of *Section 11.5, Corrigendum 5 Sequences*, as follows:

From:	first_character	intervening_character(s)	*last_character*
To:	first_character	*last_character*	intervening_character(s)

   - Replace the first_character and last_character sequence with the canonically equivalent composed character, according to the Canonical Composition Algorithm.

Note: For step 3, in most implementations it is actually more efficient (and much simpler) to parameterize the code to provide for both pre- and post-Unicode 4.1 behavior. This typically takes only one additional conditional statement.

Implementations of the Unicode Normalization Algorithm prior to version 4.1 were not all consistent with each other. Some followed the letter of the specification; because of the defect in the specification addressed by Corrigendum #5 [Corrigendum5], such implementations

were not idempotent, and their normalization results for the edge cases addressed by the corrigendum were not always well-defined. Other implementations followed the intent of the specification and implemented based on the normalization examples and reference code; those implementations behave *as if* Corrigendum #5 had already been applied. When developing a current implementation to guarantee process stability even for earlier versions of the standard, it is important to know which type of earlier Unicode implementation of normalization is being targeted. Step 3 outlined above only needs to be applied to guarantee process stability for interoperating with early implementations that followed the letter of the specification prior to version 4.1. Step 3 can be omitted when interoperating with implementations that behaved as if Corrigendum #5 had already been applied.

11.4 Forbidding Characters

An alternative approach for certain protocols is to forbid characters that differ in normalization status across versions prior to Unicode 4.1. The characters and sequences affected are not in any practical use, so this may be viable for some implementations. For example, when upgrading from Unicode 3.2 to Unicode 5.0, there are three relevant corrigenda:

- Corrigendum #3, "U+F951 Normalization" [Corrigendum3]
- Corrigendum #4, "Five Unihan Canonical Mapping Errors" [Corrigendum4]
  The five characters are U+2F868, U+2F874, U+2F91F, U+2F95F, and U+2F9BF.
- Corrigendum #5, "Normalization Idempotency" [Corrigendum5]

The characters in Corrigenda #3 and #4 are all extremely rare Han characters. They are compatibility characters included only for compatibility with a single East Asian character set standard each: U+F951 for a duplicate character in KS X 1001, and the other five for CNS 11643-1992. That's why they have canonical decomposition mappings in the first place.

The duplicate character in KS X 1001 is a rare character in Korean to begin with—in a South Korean standard, where the use of Han characters at all is uncommon in actual data. And this is a pronunciation duplicate, which even if it were used would very likely be inconsistently and incorrectly used by end users, because there is no visual way for them to make the correct distinctions.

The five characters from CNS 11643-1992 have even less utility. They are minor glyphic variants of unified characters—the kinds of distinctions that are subsumed already within all the unified Han ideographs in the Unicode Standard. They are from Planes 4–15 of CNS 11643-1992, which never saw any commercial implementation in Taiwan. The IT systems in Taiwan almost all implemented Big Five instead, which was a slight variant on Planes 1 and 2 of CNS 11643-1986, and which included none of the five glyph variants in question here.

As for Corrigendum #5, it is important to recognize that none of the affected sequences occur in any well-formed text in any language. See *Section 11.5, Corrigendum 5 Sequences.*

11.5 Corrigendum 5 Sequences

Table 10 shows all of the problem sequences relevant to Corrigendum 5. *It is important to emphasize that none of these sequences will occur in any meaningful text, because none of the intervening characters shown in the sequences occur in the contexts shown in the table.*

Table 10. Problem Sequences

First Character	Intervening Character(s)	Last Character
09C7 BENGALI VOWEL SIGN E	One or more characters with a non-zero Canonical Combining Class property value — for example, an acute accent.	09BE BENGALI VOWEL SIGN AA or 09D7 BENGALI AU LENGTH MARK
0B47 ORIYA VOWEL SIGN E		0B3E ORIYA VOWEL SIGN AA or 0B56 ORIYA AI LENGTH MARK or 0B57 ORIYA AU LENGTH MARK
0BC6 TAMIL VOWEL SIGN E		0BBE TAMIL VOWEL SIGN AA or 0BD7 TAMIL AU LENGTH MARK
0BC7 TAMIL VOWEL SIGN EE		0BBE TAMIL VOWEL SIGN AA
0B92 TAMIL LETTER O		0BD7 TAMIL AU LENGTH MARK
0CC6 KANNADA VOWEL SIGN E		0CC2 KANNADA VOWEL SIGN UU or 0CD5 KANNADA LENGTH MARK or 0CD6 KANNADA AI LENGTH MARK
0CBF KANNADA VOWEL SIGN I or 0CCA KANNADA VOWEL SIGN O		0CD5 KANNADA LENGTH MARK
0D47 MALAYALAM VOWEL SIGN EE		0D3E MALAYALAM VOWEL SIGN AA
0D46 MALAYALAM VOWEL SIGN E		0D3E MALAYALAM VOWEL SIGN AA or 0D57 MALAYALAM AU LENGTH MARK
1025 MYANMAR LETTER U		102E MYANMAR VOWEL SIGN II
0DD9 SINHALA VOWEL SIGN KOMBUVA		0DCF SINHALA VOWEL SIGN AELA-PILLA or 0DDF SINHALA VOWEL SIGN GAYANUKITTA
[1100–1112] HANGUL CHOSEONG KIYEOK..HIEUH (19 instances)		[1161–1175] HANGUL JUNGSEONG A..I (21 instances)
[:HangulSyllableType=LV:]		[11A8..11C2] HANGUL JONGSEONG KIYEOK..HIEUH

	(27 instances)

Note: This table is constructed on the premise that the text is being normalized and that the first character has already been composed if possible. If the table is used externally to normalization to assess whether any problem sequences occur, then the implementation must also catch cases that are canonical equivalents. That is only relevant to the case [:HangulSyllableType=LV:]; the equivalent sequences of <x,y> where x is in [1100..1112] and y is in [1161..1175] must also be detected.

12 Stabilized Strings

In certain protocols, there is a requirement for a normalization process for *stabilized* strings. The key concept is that for a given normalization form, once a Unicode string has been successfully normalized according to the process, it will *never* change if subsequently normalized again, in any version of Unicode, past or future. To meet this need, the *Normalization Process for Stabilized Strings* (NPSS) is defined. NPSS adds to regular normalization the requirement that an implementation must abort with an error if it encounters any characters that are not in the current version of Unicode.

12.1 Normalization Process for Stabilized Strings

The Normalization Process for Stabilized Strings (NPSS) for a given normalization form (NFD, NFC, NFKD, or NFKC) is the same as the corresponding process for generating that form, except that:

- The process must be aborted with an error if the string contains any code point with the property value General_Category=Unassigned, according to the version of Unicode used for the normalization process.

Examples:

Sample Characters	Required Behavior for Unicode Version			
	3.2	4.0	4.1	5.0
U+0234 (ȴ) LATIN SMALL LETTER L WITH CURL (added in Unicode 4.0)	Abort	Accept	Accept	Accept
U+0237 (ȷ) LATIN SMALL LETTER DOTLESS J (added in Unicode 4.1)	Abort	Abort	Accept	Accept
U+0242 (ɂ) LATIN SMALL LETTER GLOTTAL STOP (added in Unicode 5.0)	Abort	Abort	Abort	Accept

Once a string has been normalized by the NPSS for a particular normalization form, it will never change if renormalized for that same normalization form by an implementation that supports any version of Unicode, past or future. For example, if an implementation normalizes a string to NFC, following the constraints of NPSS (aborting with an error if it encounters any unassigned code point for the version of Unicode it supports), the resulting normalized string would be stable: it would remain completely unchanged if renormalized to NFC by any

conformant Unicode normalization implementation supporting a prior or a future version of the standard.

Note that NPSS defines a process, not another normalization form. The resulting string is simply in a particular normalization form. If a different implementation applies the NPSS again to that string, then depending on the version of Unicode supported by the other implementation, either the same string will result, or an error will occur. Given a string that is purported to have been produced by the NPSS for a given normalization form, what an implementation can determine is one of the following three conditions:

1. definitely produced by NPSS (it is normalized, and contains no unassigned characters)
2. definitely not produced by NPSS (it is not normalized)
3. may or may not have been produced by NPSS (it contains unassigned characters but is otherwise normalized)

The additional data required for the stable normalization process can be easily implemented with a compact lookup table. Most libraries supplying normalization functions also supply the required property tests, and in those normalization functions it is straightforward for them to provide an additional parameter which invokes the stabilized process.

NPSS only applies to Unicode 4.1 and later, or to implementations that apply Corrigenda #2 through #5 to earlier versions: see *Section 11 Stability Prior to Unicode 4.1*. A protocol that requires stability even across other versions is a *restricted* protocol. Such a protocol must define and use a *restricted* NPSS, a process that also aborts with an error if encounters any problematic characters or sequences, as discussed in *Section 11.4 Forbidding Characters*.

13 Stream-Safe Text Format

There are certain protocols that would benefit from using normalization, but that have implementation constraints. For example, a protocol may require buffered serialization, in which only a portion of a string may be available at a given time. Consider the extreme case of a string containing a *digit 2* followed by 10,000 *umlauts* followed by one *dot-below*, then a *digit 3*. As part of normalization, the *dot-below* at the end must be reordered to immediately after the *digit 2*, which means that 10,003 characters need to be considered before the result can be output.

Such extremely long sequences of combining marks are not illegal, even though for all practical purposes they are not meaningful. However, the possibility of encountering such sequences forces a conformant, serializing implementation to provide large buffer capacity or to provide a special exception mechanism just for such degenerate cases. The Stream-Safe Text Format specification addresses this situation.

UAX15-D3. *Stream-Safe Text Format:* A Unicode string is said to be in Stream-Safe Text Format if it would not contain any sequences of non-starters longer than 30 characters in length when normalized to NFKD.

- Such a string can be normalized in buffered serialization with a buffer size of 32 characters, which would require no more than 128 bytes in any Unicode Encoding Form.
- Incorrect buffer handling can introduce subtle errors in the results. Any buffered implementation should be carefully checked against the normalization test data.
- The value of 30 is chosen to be significantly beyond what is required for any linguistic or technical usage. While it would have been feasible to chose a smaller number, this value provides a very wide margin, yet is well within the buffer size limits of practical implementations.

- NFKD was chosen for the definition because it produces the potentially longest sequences of non-starters from the same text.

UAX15-D4. *Stream-Safe Text Process* is the process of producing a Unicode string in Stream-Safe Text Format by processing that string from start to finish, inserting U+034F COMBINING GRAPHEME JOINER (CGJ) within long sequences of non-starters. The exact position of the inserted CGJs are determined according to the following algorithm, which describes the generation of an output string from an input string:

1. If the input string is empty, return an empty output string.
2. Set nonStarterCount to zero.
3. For each code point C in the input string:
   a. Produce the NFKD decomposition S.
   b. If nonStarterCount plus the number of initial non-starters in S is greater than 30, append a CGJ to the output string and set the nonStarterCount to zero.
   c. Append C to the output string.
   d. If there are no starters in S, increment nonStarterCount by the number of code points in S; otherwise, set nonStarterCount to the number of trailing non-starters in S (which may be zero).
4. Return the output string.

The Stream-Safe Text Process ensures not only that the resulting text is in Stream-Safe Text Format, but that any normalization of the result is also in Stream-Safe Text Format. This is true for any input string that does not contain unassigned code points. The Stream-Safe Text Process preserves all of the four normalization forms defined in this annex (NFC, NFD, NFKC, NFKD). However, normalization and the Stream-Safe Text Process do not commute. That is, normalizing an arbitrary text to NFC, followed by applying the Stream-Safe Text Process, is not guaranteed to produce the same result as applying the Stream-Safe Text Process to that arbitrary text, followed by normalization to NFC.

It is important to realize that if the Stream-Safe Text Process does modify the input text by insertion of CGJs, the result will *not* be canonically equivalent to the original. The Stream-Safe Text Format is designed for use in protocols and systems that accept the limitations on the text imposed by the format, just as they may impose their own limitations, such as removing certain control codes.

However, the Stream-Safe Text Format will not modify ordinary texts. Where it modifies an exceptional text, the resulting string would no longer be canonically equivalent to the original, but the modifications are minor and do not disturb any meaningful content. The modified text contains all of the content of the original, with the only difference being that reordering is blocked across long groups of non-starters. Any text in Stream-Safe Text Format can be normalized with very small buffers using any of the standard Normalization Forms.

Implementations can optimize this specification as long as they produce the same results. In particular, the information used in Step 3 can be precomputed: it does not require the actual normalization of the character. For efficient processing, the Stream-Safe Text Process can be implemented in the same implementation pass as normalization. In such a case, the choice of whether to apply the Stream-Safe Text Process can be controlled by an input parameter.

13.1 Buffering with Unicode Normalization

Using buffers for normalization requires that characters be emptied from the buffer correctly. That is, as decompositions are appended to the buffer, periodically the end of the buffer will be reached. At that time, the characters in the buffer up to *but not including* the last character with the property value Quick_Check=Yes (QC=Y) must be canonically ordered (and if NFC

and NFKC are being generated, must also be composed), and only then flushed. For more information on the Quick_Check property, see *Section 9 Detecting Normalization Forms*.

Consider the following example. Text is being normalized into NFC with a buffer size of 40. The buffer has been successively filled with decompositions, and has two remaining slots. The decomposition takes three characters, and wouldn't fit. The last character with QC=Y is the "s", highlighted in color below.

Buffer

T	h	e		c	◌́	a	…	p	◌̃	q	r	◌́	s	◌́		
0	1	2	3	4	5	6	…	31	32	33	34	35	36	37	38	39

Decomposition

u	◌̃	◌́
0	1	2

Thus the buffer up to but not including "s" needs to be composed, and flushed. Once this is done, the decomposition can be appended, and the buffer is left in the following state:

s	◌́	u	◌̃	◌́			…									
0	1	2	3	4	5	6	…	31	32	33	34	35	36	37	38	39

Implementations may also canonically order (and compose) the contents of the buffer as they go; the key requirement is that they cannot compose a sequence until a following character with the property QC=Y is encountered. For example, if that had been done in the above example, then during the course of filling the buffer, we would have had the following state, where "c" is the last character with QC=Y.

T	h	e		c	◌́											
0	1	2	3	4	5	6	…	31	32	33	34	35	36	37	38	39

When the "a" (with QC=Y) is to be appended to the buffer, it is then safe to compose the "c" and all subsequent characters, and then enter in the "a", marking it as the last character with QC=Y.

T	h	e		ć	a											
0	1	2	3	4	5	6	…	31	32	33	34	35	36	37	38	39

14 Implementation Notes

14.1 Optimization Strategies

There are a number of optimizations that can be made in programs that normalize Unicode strings. This section lists a few techniques for optimization. See also [UTN5] for other

information about possible optimizations.

Any implementation using optimization techniques must be carefully checked to ensure that it still produces conformant results. In particular, the code must still be able to pass the the NormalizationTest.txt conformance test [Tests15].

14.1.1 Quick Check for NFC

When normalizing to NFC, rather than first decomposing the text fully, a quick check can be made on each character. If it is already in the proper precomposed form, then no work has to be done. Only if the current character is a combining mark or is in the Composition Exclusion Table [Exclusions], does a slower code path need to be invoked. The slower code path will need to look at previous characters, back to the last starter. See *Section 9, Detecting Normalization Forms*, for more information.

14.1.2 Optimizing Tables for NFC Composition

The majority of the cycles spent in doing composition are spent looking up the appropriate data. The data lookup for Normalization Form C can be very efficiently implemented, because it has to look up only pairs of characters, rather than arbitrary strings. First, a multistage table (also known as a *trie;* see *Chapter 5, Implementation Guidelines* in [Unicode]) is used to map a character *c* to a small integer *i* in a contiguous range from 0 to *n*. The code for doing this looks like:

```
i = data[index[c >> BLOCKSHIFT] + (c & BLOCKMASK)];
```

Then a pair of these small integers are simply mapped through a two-dimensional array to get a resulting value. This yields much better performance than a general-purpose string lookup in a hash table.

14.1.3 Optimizing Tables for NFD Quick Check

The values of the Canonical_Combining_Class property are constrained by the character encoding stability guarantees to the range 0..254; the value 255 will never be assigned for a Canonical_Combining_Class value. Because of this constraint, implementations can make use of 255 as an implementation-specific value for optimizing data tables. For example, one can do a fast and compact table for implementing isNFD(x) by using the value 255 to represent NFKC_QC=No.

14.1.4 Hangul Decomposition and Composition

Because the decompositions and compositions for Hangul syllables are algorithmic, memory storage can be significantly reduced if the corresponding operations are done in code, rather than by simply storing the data in the general-purpose tables. See *Section 3.12, Combining Jamo Behavior* in [Unicode] for example code illustrating the Hangul Syllable Decomposition and the Hangul Syllable Composition algorithms.

14.2 Code Samples

Perl code implementing normalization is available on the W3C site [CharLint].

See also the [FAQ] pages regarding normalization for pointers to demonstrations of normalization sample code.

Appendix A: Intellectual Property Considerations

Transcript of letter regarding disclosure of IBM Technology
(Hard copy is on file with the Chair of UTC and the Chair of NCITS/L2)
Transcribed on 1999-03-10

February 26, 1999

The Chair, Unicode Technical Committee

Subject: Disclosure of IBM Technology - Unicode Normalization Forms

The attached document entitled "Unicode Normalization Forms" does not require IBM technology, but may be implemented using IBM technology that has been filed for US Patent. However, IBM believes that the technology could be beneficial to the software community at large, especially with respect to usage on the Internet, allowing the community to derive the enormous benefits provided by Unicode.

This letter is to inform you that IBM is pleased to make the Unicode normalization technology that has been filed for patent freely available to anyone using them in implementing to the Unicode standard.

Sincerely,

W. J. Sullivan,
Acting Director of National Language Support
and Information Development

Acknowledgments

Mark Davis and Martin Dürst created the initial versions of this annex. Mark Davis added to the text through Unicode 5.1. Ken Whistler has maintained the text since Unicode 5.2.

Thanks to Kent Karlsson, Marcin Kowalczyk, Rick Kunst, Per Mildner, Terry Reedy, Sadahiro Tomoyuki, Markus Scherer, Dick Sites, Ienup Sung, and Erik van der Poel for feedback on this annex, including earlier versions. Asmus Freytag extensively reformatted the text for publication as part of the Unicode 5.0 book.

References

For references for this annex, see Unicode Standard Annex #41, "Common References for Unicode Standard Annexes."

Modifications

The following summarizes modifications from the previous version of this annex.

Revision 49 [KW]

- **Proposed Update** for Version 13.0.0.
- Updated explanation of script-specific exclusions in *Section 5.1, Composition Exclusion Types*.

Previous revisions can be accessed with the "Previous Version" link in the header.