Internet Draft
<draft-duerst-i18n-norm-04.txt>
Expires in six months

M. Duerst
W3C/Keio University
M. Davis
IBM

September 2000

Character Normalization in IETF Protocols

Status of this Memo

This document is not a product of any working group, but may be discussed on the mailing lists <www-international@w3.org> or <discuss@apps.ietf.org>.

Abstract

The Universal Character Set (UCS) [ISO10646, Unicode] covers a very wide repertoire of characters. The IETF, in [RFC 2277], requires that future IETF protocols support UTF-8 [RFC 2279], an ASCII-compatible encoding of UCS. The wide range of characters included in the UCS has lead to some cases of duplicate encodings. This document proposes that in IETF protocols, the class of duplicates called canonical equivalents be dealt with by using Early Uniform Normalization according to Unicode Normalization Form C, Canonical Composition (NFC) [UTR15]. This document describes both Early Uniform Normalization and Normalization Form C.

Table of contents

0. Change Log

Changes from -03 to -04

- Changed intro to make clear this is mainly about canonical equivalences
- Made UTR#15, V18.0, the normative description of NFC
- Added subsection on interaction with text processing (3.4.11)
- Added various examples
- Various small wording changes
- Added reference to test file
- Added a note re. terminology (Normalization vs. Canonicalization)

Changes from -02 to -03

- Fixed a bad typo in the title.
- Made a lot of wording corrections and presentation improvements, most of them suggested by Paul Hoffman.

1. Introduction

1.1 Motivation

The Universal Character Set (UCS) [ISO10646, Unicode] covers a very wide repertoire of characters. The IETF, in [RFC 2277], requires that future IETF protocols support UTF-8 [RFC 2279], an ASCII-compatible encoding of UCS. The need for round-trip convertion to pre-existing character encodings has led to some cases of duplicate encodings. This has lead to uncertainity for protocol specifiers and implementers, because it was not clear which part of the Internet infrastructure should take responsibility for these duplicates, and how.

Besides straight-out duplicates, there are also many cases of characters that are in one way or another similar. The equivalence between duplicates is called canonical equivalence. Many of the equivalences between similar characters are called compatibility equivalences. This document concentrates on canonical equivalence. The various cases of similar characters are listed in Section 5.

There are mainly two kinds of canonical equivalences, singleton equivalences and precomposed/decomposed equivalences. Both of these can be illustrated using the character A with a ring above. This character can be encoded in three ways:

1) U+00C5 LATIN CAPITAL LETTER A WITH RING ABOVE
2) U+0041 LATIN CAPITAL LETTER A followed by U+030A COMBINING RING ABOVE
3) U+212B ANGSTROM SIGN

The equivalence between 1) and 3) is a singleton equivalence. The equivalence between 1) and 2) is a precomposed/decomposed equivalence, where 1) is the precomposed representation, and 2) is the decomposed representation.

In all three cases, it is supposed to look the same for the reader. Also, applications may use one or another representation, or even more than one, but they are not allowed to assume that other applications will preserve the difference between them.

The inclusion of these various representation alternatives was a result of the requirement for round trip conversion with a wide range of legacy encodings as well as of the merger between Unicode and ISO 10646.

The Unicode Standard from early on has defined Canonical Equivalence to make clear which sequences of codepoints cases should be treated as pure encoding duplicates and which sequences of codepoints should be treated as genuinely different (if maybe in some cases closely related) data. The Unicode Standard also from early on defined decomposed normalization, what is now called Normalization Form D (case 2) in the example above). This is very well suited for some kinds of internal processing, but decomposition does not correspond to how data gets converted from legacy encodings and transmitted on the Internet. In that case, precomposed data (i.e. case 1) in the example above) is prevalent.

Note: This specification uses the term 'codepoint', and not 'character', to make clear that it speaks about what the standards encode, and not what the end users think about, which is not always the same.

Encouraged by many factors such as a requirements analysis of the W3C [Charreq], the Unicode Technical Committee defined Normalization

Form C, Canonical Composition (see [UTR15]). Normalization Form C in general produces the same representation as straightforward transcoding from legacy encodings (See Section 3.4 for the known exception). The careful and detailled definition of Normalization Form C is mainly needed to unambigously define edge cases (base letters with two or more combining characters). Most of these edge cases will turn up extremely rarely in actual data.

The W3C is adapting Normalization Form C in the form of Early Uniform Normalization, which means that it assumes that in general, data will be already in Normalization Form C [Charmod].

This document recommends that in IETF protocols, Canonical Equivalents be dealt with by using Early Uniform Normalization according to Unicode Normalization Form C, Canonical Composition [UTR15]. This document describes both Early Uniform Normalization (in Section 2) and Normalization Form C (in Section 3). Section 4 contains an analysis of (mostly theoretical) potential risks for the stability of Normalization Form C. For reference, Section 5 discusses various cases of equivalences not dealt with by Normalization Form C.

Note: The terms 'normalization' (such as in 'Normalization Form C') and 'canonicalization' (such as in XML Canonicalization) can mean virtually the same thing. In the context of the topics described in this document, only 'normalization' is used because 'canonical' is used to distinguish between canonical equivalents and compatibility equivalents.

1.2 Notational Conventions

For UCS codepoints, the notation U+HHHH is used, where HHHH is the hexadecimal representation of the codepoint. This may be followed by the official name of the character in all caps.

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this specification are to be interpreted as described in [RFC2119].

2. Early Uniform Normalization

This section tries to give some guidance on how Normalization Form C (NFC), described later in Section 3, should be used by Internet protocols. Each Internet protocol has to define by itself how to use NFC, and has to take into account its particular needs. However, the advice in this section is intended to help writers of specifications not very familliar with text normalization issues, and to try to make sure that the various protocols use solutions that interface easily with each other.

This section uses various well-known Internet protocols as examples. However, such examples do not imply that the protocol elements mentioned actually accept non-ASCII characters. Depending on the protocol element mentioned, that may or may not be the case, and may change in the future. Also, the examples are not intended to actually define how a specific protocol deals with text normalization issues. This is the responsibility of the specification for each specific protocol.

The basic principle for how to use Normalization Form C is Early Uniform Normalization. This means that ideally, only text in Normalization Form C appears on the wire on the Internet. This can be seen as applying 'be conservative in what you send' to the problem of text normalization. And (again ideally) it should not be needed that each implemenation of an Internet protocol separately implements normalization. Text should just be provided normalized from the underlying infrastructure, e.g. the operating system or the keyboard driver.

Early normalization is of particular importance for those parts of Internet protocols that are used as identifiers. Examples would be URIs, domain names, email addresses, identifier names in PKIX certificates, identifiers in ACAP, file names in FTP, folder names in IMAP, newsgroup names in NNTP, and so on. This is due to the following reasons:

- In order for the protocol to work, it has to be very well defined when two protocol element values match and when not.
- Implementations, in particular on the server side, do not in any way have to deal with e.g. display of multilingual text, but on the other hand have to handle a lot of protocol-specific issues. Such implementations therefore should not be bothered with text normalization.

For free text, e.g. the content of mail messages or news postings, Early Uniform Normalization is somewhat less important, but definitely improves interoperability.

For protocol elements used as identifiers, this document recommends Internet protocols to specify the following:

- Comparison SHOULD be carried out purely binary (after it has been made sure, where necessary, that the texts to be compared are in the same character encoding).
- Any kind of text, and in particular identifier-like protocol elements, SHOULD be sent normalized to Normalization Form C.
- In case comparison fails due to a difference in text normalization, the originator of the non-normalized text is responsible for the failure.
- In case implementors are aware of the fact, or suspect, that their underlying infrastructure produces non-normalized text, they SHOULD take care to do the necessary tests, and if necessary the actual normalization, by themselves.

- In the case of creation of identifiers, and in particular if this creation is comparatively infrequent (e.g. newsgroup names, domain names), and happens in a rather centralized manner, explicit checks for normalization SHOULD be required by the protocol specification.

3. Canonical Composition (Normalization Form C)

This section describes Canonical Composition (Normalization Form C, NFC). The normative specification of Canonical Composition is found in [UTR15]. The description is done in a procedural way, but any other procedure that leads to identical results can be used. The result is supposed to be exactly identical to that described by [UTR15]. If any differences should be found, [UTR15] must be followed. For each step, various notes are provided to help understand the description and give implementation hints.

Given a sequence of UCS codepoints, its Canonical Composition can be computed with the following three steps:

1. **Decomposition** (Section 3.1)
2. **Reordering** (Section 3.2)
3. **Recomposition** (Section 3.3)

Additional implementation notes are given in Section 3.4.

3.1 Decomposition

For each UCS codepoint in the input sequence, check whether this codepoint has a canonical decomposition according to the newest version of the Unicode Character Database (field 5 in [UniData]). If such a decomposition is found, replace the codepoint in the input sequence by the codepoint(s) in the decomposition, and recursivly check for and apply decomposition on the first replaced codepoint.

Note: Fields in [UniData] are delimited by ';'. Field 5 in [UniData] is the 6th field when counting with an index origin of 1. Fields starting with a tag delimited by '<' and '>' indicate compatibility decompositions; these compatibility decompositions MUST NOT be used for Normalization Form C.

Note: For Korean Hangul, the decompositions are not contained in [UniData], but have to be generated algorithmically according to the description in [Unicode], Section 3.11.

Note: Some decompositions replace a single codepoint by another single codepoint.

Note: It is not necessary to check replaced codepoints other than the first one due to the properties of the data in the Unicode Character Database.

Note: It is possible to 'precompile' the decompositions to avoid having to apply them recursively.

3.2 Reordering

For each adjacent pair of UCS codepoints after decomposition, check the combining classes of the UCS codepoints according to the newest version of the Unicode Character Database (Field 3 in [UniData]). If the combining class of the first codepoint is higher than the combining class of the second codepoint, and at the same time the combining class of the second codepoint is not zero, then exchange the two codepoints. Repeat this process until no two codepoints can be exchanged anymore.

Note: A combining class greater than zero indicates that a codepoint is a combining mark that participates in reordering. A combining class of zero indicates that a codepoint is not a combining mark, or that it is a combining mark that is not affected by reordering. There are no combining classes below zero.

Note: Besides a few script-specific combining classes, combining classes mainly distinguish whether a combining mark is attached to the base letter or just placed near the base letter, and on which side of the base letter (e.g. bottom, above right,...) the combining mark is attached/placed. Reordering assures that combining marks placed on different sides of the same character are placed in a canonical order (because any order would visually look the same), while combining marks placed on the same side of a character are not reordered (because reordering them would change the combination they represent).

Note: After completing this step, the sequence of UCS codepoints is in Canonical Decomposition (Normalization Form D).

3.3 Recomposition

This section describes recomposition in a top-down manner, first describing recomposition processing in general (Section 3.3.1), then describing which pairs of codepoints can be canonically combined (Section 3.3.2) and then describing the combination exclusions.

3.3.1 Recomposition Processing

Process the sequence of UCS codepoints resulting from Reordering from start to end. This process requires a state variable called 'initial'. At the beginning of the process, the value of 'initial' is empty.

```
For each codepoint in the sequence resulting from Reordering,
do the following:
- If the following three conditions all apply
   - 'initial' has a value
   - the codepoint immediately preceeding the current codepoint
     is this 'initial' or has a combining class not equal to the
     combining class of the current codepoint
   - the 'initial' can be canonically recombined (see Section 3.3.1)
     with with the current codepoint
   then replace the 'initial' with the canonical recombination and
   remove the current codepoint.
- Otherwise, if the current codepoint has combining class zero,
   store its value in 'initial'.

Note: At the beginning of recomposition, there is no 'initial'.
    An 'initial' is remembered as soon as the first codepoint
    with a combining class of zero is found. Not every codepoint
    with a combining class of zero becomes an 'initial'; the
    exceptions are those that are the second codepoint in
    a recomposition. The 'initial' as used in this description
    is slightly different from the 'starter' as defined in [UTR15],
    but this does not affect the result.

Note: Checking the previous codepoint to have a combining class
    smaller than the combining class of the current codepoint
    (except if the previous codepoint is the 'initial' and therefore
    has a combining class of zero) assures that the conditions used
    for reordering are maintained in the recombination step.

Note: Other algorithms for recomposition have been considered, but
    this algorithm has been choosen because it provides a very good
    balance between computational and implementation complexity
    and 'power' of recombination. As an example, assume a text contains
    a U+0041 LATIN CAPITAL LETTER A with a U+030A COMBINING RING ABOVE
    and a U+031F COMBINING PLUS SIGN BELOW. Because the canonical
    reordering puts the COMBINING PLUS SIGN BELOW before the COMBINING
    RING ABOVE, a more straightforward algorithm would not be able
    to recombine this to U+00C5 LATIN CAPITAL LETTER A WITH RING ABOVE
    followed by U+031F COMBINING PLUS SIGN BELOW.
```

3.3.2 Pairs of Codepoints that can be Canonically Recombined

A pair of codepoints can be canonically recombined to a third codepoint if this third codepoint has a canonical decomposition into the sequence of two codepoints (see [UniData], field 5) and this canonical decomposition is not excluded from recombination. For Korean Hangul, the redecompositions are not contained in [UniData], but have to be generated algorithmically according to the description in [Unicode], Section 3.11.

3.3.3 Combination Exclusions

The exclusions from recombination are defined as follows:

1) Singletons: Codepoints that have a canonical decomposition into a single other codepoint (example: U+212B ANGSTROM SIGN).
2) Non-starter: A codepoint with a decomposition starting with a codepoint of a combining class other than zero (example: U+0F75 TIBETAN VOWEL SIGN UU).
3) Post-Unicode3.0: A codepoint with a decomposition introduced after Unicode 3.0 (no applicable example).
4) Script-specific: Precomposed codepoints that are not the generally preferred form for their script (example: U+0959 DEVANAGARI LETTER KHHA).

The list of codepoints for 1) and 2) can be produced directly from the Unicode Character Database [UniData]. The list of codepoints for 3) can be produced from a comparison between the 3.0.0 version and the latest version of [UniData], but this may be difficult. The list of codepoints for 4) cannot be computed. For 3) and 4), the lists provided in [CompExcl] MUST be used. [CompExcl] also provides lists for 1) and 2) for cross-checking. The list for 3) is currently empty because there are at the moment no post-Unicode3.0 codepoints with decompositions.

Note: Exclusion of singletons is necessary because in a pair of canonically equivalent codepoints, the canonical decomposition points from the 'less desirable' codepoint to the preferred codepoint. In this case, both canonical decomposition and canonical composition have the same preference.

Note: For discussion of the exclusion of Post-Unicode3.0 codepoints from recombination, please see Section 4 on versioning issues.

3.4 Implementation Notes

This section contains various notes on potential implementation issues, improvements, and shortcuts. Further notes on implementation may be found in [UTR15] or in newer versions of that document.

3.4.1 Avoiding Decomposition, and Checking for Normalization Form C

It is not always necessary to decompose and recompose. In particular, any sequence that does not contain any of the following is already in Normalization Form C:

- Codepoints that are excluded from recomposition (see Section 3.3.3)
- Codepoints that appear in second position in a canonical recomposition
- Hangul Jamo codepoints (U+1100-U+11F9)

- Unassigned codepoints

If a contiguous part of a sequence satisfies the above criterion all but the last of the codepoints are already in Normalization Form C.

The above criteria can also be used to easily check that some data is already in Normalization Form C. However, this check will reject some cases that actually are normalized.

3.4.2 Unassigned Codepoints

Unassigned codepoints (codepoints that are not assigned in the current version of Unicode) are listed above to avoid claiming that something is in Normalization Form C when it may indeed not be, but they usually will be treated differently from others. The following behaviours may be possible, depending on the context of normalization:

- Stop the normalization process with a fatal error. (This should be done only in very exceptional circumstances. It would mean that the implementation will die with data that conforms to a future version of Unicode.)
- Produce some warning that such codepoints have been seen, for further checking.
- Just copy the unassigned codepoint from the input to the output, running the risk of not normalizing completely.
- Checking that the program-internal data is up to date via the Internet.
- Distinguish behaviour depending on which range of codepoints the unassigned codepoint has been found

3.4.3 Surrogates

When implementing normalization for sequences of UCS codepoints represented as UTF-16 code units, care has to be taken that pairs of surrogate code units that represent a single UCS codepoint are treated appropriately.

3.4.4 Korean Hangul

There are no interactions between normalization of Korean Hangul and the other normalizations. These two parts of normalization can therefore be carried out separately, with different implementation improvements.

3.4.5 Piecewise Application

The various steps such as decomposition, reordering, and recomposition, can be applied to appropriately choosen parts of a codepoint sequence. As an example, when normalizing a large file, normalization can be done on each line separately because line endings and normalization do not interact.

3.4.6 Integrating Decomposition and Recomposition

It is possible to avoid full decomposition by noting that decomposition of a codepoint that is not in the exclusion list can be avoided if it is not followed by a codepoint that can appear in second position in a canonical recomposition. This condition can be strengthened by noting that decomposition is not necessary if the combining class of the following codepoint is higher than the highest combining class obtained from decomposing the character in question. In other cases, a decomposition followed immediately by a recomposition can be precalculated. Further details are left to the reader.

3.4.7 Decomposition

Recursive application of decomposition can be avoided by a preprocessing step that calculates a full canonical decomposition for each character with a canonical decomposition.

3.4.8 Reordering

The reordering step basically is a sorting problem. Because the number of consecutive combining marks (i.e. consecutive codepoints with combining class greater than zero) is usually extremely small, a very simple sorting algorithm can be used, e.g. a straightforward bubble sort.

Because reordering will occur extremely locally, the following variant of bubble sort will lead to a fast and simple implementation:

- Start checking the first pair (e.g. the first two codepoints).
- If there is an exchange, and we are not at the start of the sequence, move back by one codepoint and check again.
- Otherwise (i.e. if there is no exchange, or we are at the start of the sequence) and we are not at the end of the sequence, move forward by one codepoint and check again.
- If we are at the end of the sequence, and there has been no exchange for the last pair, then we are done.

3.4.9 Conversion from Legacy Encodings

Normalization Form C is designed so that in almost all cases, one-to-one conversion from legacy encodings (e.g. iso-8859-1,...) to UCS will produce a result that is already in Normalization Form C.

The one exception to this known at the moment is code page 1252 (charset=windows-1258, for Vietnamese, [windows-1258]). This character encoding uses a kind of 'half-precomposed' encoding, whereas Normalization Form C uses full precomposition for the characters needed for Vietnamese. As an example, U+1EAD LATIN SMALL LETTER A WITH CIRCUMFLEX AND DOT BELOW is encoded as U+00E2 LATIN SMALL LETTER A WITH CIRCUMFLEX followed by U+0323 COMBINING DOT BELOW in code page 1252, but U+1EAD is the normalized form.

3.4.10 Uses of UCS in Non-Normalized Form

One known case where the UCS is used in a way that is not in Normalization Form C is a group of users using the UCS for Yiddish. The few combinations of Hebrew base letters and diacritics used to write Yiddish are available precomposed in UCS (example: U+FB2F HEBREW LETTER ALEF WITH QAMATS). On the other hand, the many combinations used in writing the Hebrew language are only available by using combining characters.

In order to lead to an uniform model of encoding Hebrew, the precomposed Hebrew codepoints were excluded from recombination. This means that Yiddish using precomposed codepoints is not in Normalization Form C.

3.4.11 Interaction with Text Processing

There are many operations on text strings that can create non-normalized output even if the input was normalized. Examples are concatenation (if the second string starts with one of the characters discussed is Section 3.4.1) or case changes (as an example, 1E98 LATIN SMALL LETTER W WITH RING ABOVE does not have a precomposed capital equivalent).

3.4.12 Implementations and Test Suites

Implementation examples can be found at [Charlint] (Perl), [ICU] (C/C++) and [Normalizer] (Java).

A huge file with test cases for normalization is avaliable as part of Unicode 3.0.1 [NormTest].

4. Stability and Versioning

Defining a normalization form for Internet-wide use requires that this normalization form stays as stable as possible. Stability for Normalization Form C is mainly achieved by introducing a cutoff version. For precomposed characters encoded up to and including this version, in principle the precomposed version is the normal form, but precompomposed codepoints introduced after the cutoff version are decomposed in Normalization Form C.

As the cutoff version, version 3.0 of Unicode and the second edition of ISO/IEC 10646-1 have been choosen. These are aligned codepoint-by-codepoint. They are both widely and integrally available, i.e. they do not reqire the application of updates ammendments.

The rest of this section discusses potential threats to the stability of Normalization Form C, the probability of such threats, and how to avoid them. [UniPolicy] documents policies adopted by the Unicode Consortium to limit the impact of changes on existing implementations.

The analysis below shows that the probability of the various threats is extremely low. The analysis is provided here to document the awareness of these treats and the measures that have to be taken to avoid them. This section is only of marginal importance to an implementer of Normalization Form C or to an author of an Internet protocol specification.

4.1 New Precomposed Codepoints

The introduction of new (post-Unicode 3.0) precomposed codepoints is not a threat to the stability of Normalization Form C. Such codepoints would just provide an alternate way of encoding characters that can already be encoded without them, by using a decomposed form. The normalization algorithm already provides for the exclusion of such characters from recomposition.

While Normalization Form C itself is not affected, such new codepoints would affect implementations of Normalization Form C, because such implementations have to be updated to correctly decompose the new codepoints.

Note: While the new codepoint may be correctly normalized only by updated implementations, once normalized neither older nor updated implementations will change anything anymore.

Because the new codepoints do not actually encode any new characters that could not be encoded before, because the new codepoints would not actually be used due to Early Uniform Normalization, and because of the above implementation problems, encoding new precomposed characters is superfluous and should be very clearly avoided.

4.2 New Combining Marks

It is in theory possible that a new combining mark would be encoded that is intended to represent decomposable pieces of already existing encoded characters. In case this indeed would happen, problems for Normalization Form C can be avoided by making sure the precomposed character that now has a decomposition is not included in the list of recoposition exclusions. While this helps for Normalization Form C, adding a canonical decomposition would affect other normalization forms, and it is therefore highly unlikely that such a canonical decomposition will ever be added in the first place.

In case new combining marks are encoded for new scripts, or in case a combining mark is introduced that does not appear in any precomposed character yet, then the appropriate normalization for these characters can easily be defined by providing the appropriate data. However, hopefully no new encoding ambiguities are introduced for new scripts.

4.3 Changed Codepoints

A major threat to the stability of Normalization Form C would come from changes to ISO/IEC 10646/Unicode itself, i.e. by moving around characters or redefining codepoint or by ISO/IEC 10646 and Unicode evolving differently in the future. These threats are not specific to Normalization Form C, but relevant for the use of the UCS in general, and are mentioned here for completeness.

Because of the very wide and increasing use of the UCS thoughout the world, the amount of resistance to any changes of defined codepoints or to any divergence between ISO/IEC 10646 and Unicode is extremely strong. Awareness about the need for stability in this point, as well as others, is particularly high due to the experiences with some changes in the early history of these standards, in particular with the reencoding of some Korean Hangul characters in ISO/IEC 10646 amendment 5 (and the corresponding change in Unicode). For the IETF in particular, the wording in [RFC 2279] and [RFC 2781] stresses the importance of stability in this respect.

5. Cases not dealt with by Canonical Equivalence

This section gives a list of cases that are not dealt with by Canonical Equivalence and Normalization Form C. This is done to help the reader understand Normalization Form C and its limits. The list in this section contains many cases of widely varying nature. In many cases, a viewer, if familiar with the script in question, will be able to distinguish the various variants.

Internet protocols can deal in various ways with the cases below. One way is to limit the characters e.g. allowed in an identifier so that all but one of the variants are disallowed. Another way is to assume that the user can make the distinction him/herself. Another is to understand that some characters or combinations of characters that would lead to confusion are very difficult to actually enter on any keyboard; it may therefore not really be worth to exclude them explicitly.

- Various ligatures (Latin, Arabic, e.g. U+FB01 LATIN SMALL LIGATURE FI vs. U+0066 LATIN SMALL LETTER F followed by U+0069 LATIN SMALL LETTER I)

- Croatian digraphs (e.g. U+01C8 LATIN CAPITAL LETTER L WITH SMALL LETTER J vs. U+004C LATIN CAPITAL LETTER L followed by U+006A LATIN SMALL LETTER J)

- Full-width Latin compatibility variants (e.g. U+FF21 FULLWIDTH LATIN CAPITAL LETTER A vs. U+0041 LATIN CAPITAL LETTER A)

- Half-width Kana and Hangul compatibility variants (e.g. U+FF76 HALFWIDTH KATAKANA LETTER KA vs. U+30AB KATAKANA LETTER KA)

- Vertical compatibility variants (U+FE35 PRESENTATION FORM FOR VERTICAL LEFT PARENTHESIS vs. U+0028 LEFT PARENTHESIS)

- Superscript/subscript variants (numbers and IPA, e.g. U+00B2 SUPERSCRIPT TWO)

- Small form compatibility variants (e.g. U+FE6A SMALL PERCENT SIGN)

- Enclosed/encircled alphanumerics, Kana, Hangul,... (e.g. U+2460 CIRCLED DIGIT ONE)

- Letterlike symbols, Roman numerals,... (e.g. U+210E PLANCK CONSTANT vs. U+0068 LATIN SMALL LETTER H)

- Squared Katakana and Latin abbreviations (units,..., e.g. U+334C SQUARE MEGATON)

- Hangul jamo representation alternatives for historical Hangul

- Presence or absence of joiner/non-joiner and other control characters

- Upper case/lower case distinction

- Distinction between Katakana and Hiragana

- Similar letters from different scripts (e.g. "A" in Latin, Greek, and Cyrillic)

- CJK ideograph variants (glyph variants introduced due to the source separation rule, simplifications)

- Various punctuation variants (apostrophes, middle dots, spaces,...)

- Ignorable whitespace, hyphens,...

- Ignorable accents,...

Many of the cases above are identified as compatibility equivalences in the Unicode database. [UTR15] defines Normalization Forms KC and KD to normalize compatibility equivalences. It may look attractive to just use Normalization Form KC instead of Normalization Form C for Internet protocols. However, while Canonical Equivalence, which forms the base of Normalization Form C, deals with a very small number of very well defined cases of complete equivalence (from an user point of view), Compatibility Equivalence comprises a very wide range of cases that usually have to be examined one at a time. If the domain of acceptable characters is suitably limited, such as for program identifiers, then NFKC may be a suitable normalization form.

6. Security Considerations

Security problems can result from:

- Improper implementations of normalization. For example, in certificate chaining, if the program validating a certificate chain mis-implements normalization rules, an attacker might be able to spoof an identity by picking a name that the validator thinks is equivalent to another name.
- The fact that normalization maps several input sequences to the same output sequence. If a digital signature calculation includes normalization, this can make it slightly easier to find a fake document that has the same digest as a real one.
- The use of normalization only in part of the applications. In particular, if software used for security purposes, e.g. to create and check digital signatures, normalizes data, but the applications actually using the data do not normalize, it can be very easy to create a fake document that can claim to be the real one but produces different behaviour.
- Different behavior in programs that do not respect canonical equivalence.

Security-related applications therefore MAY check for normalized input, but MUST NOT actually apply normalization unless is can be guaranteed that all related applications also apply normalization.

Acknowledgements

The earliest version of this Internet Draft, which dealt with quite similar issues, was entitled "Normalization of Internationalized Identifiers" and was submitted in July 1997 by the first author while he was at the University of Zurich. It benefited from ideas, advice, criticism and help from: Mark Davis, Larry Masenter, Michael Kung, Edward Cherlin, Alain LaBonte, Francois Yergeau, and others.

For the current version, the authors were encouraged in particular by Patrick Faltstrom and Paul Hoffman. The discussion of potential stability threats is based on contributions by John Cowan and Kenneth Whistler. Some security threats were pointed out by Masahiro Sekiguchi. Further contributions are due to Dan Oscarson.

References

[Charlint]   Martin Duerst. Charlint - A Character Normalization

Tool. <http://www.w3.org/International/charlint>.

[Charreq] Martin J. Duerst, Ed. Requirements for String Identity Matching and String Indexing. World Wide Web Consortium Working Draft. <http://www.w3.org/TR/WD-charreq>.

[Charmod] Martin J. Duerst and Francois Yergeau, Eds. Character Model for the World Wide Web. World Wide Web Consortium Working Draft. <http://www.w3.org/TR/charmod>.

[CompExcl] The Unicode Consortium. Composition Exclusions. <ftp://ftp.unicode.org/Public/UNIDATA/CompositionExclusions.txt>

[ICU] International Components for Unicode. <http://oss.software.ibm.com/icu/>.

[ISO10646] ISO/IEC 10646-1:2000. International standard -- Information technology -- Universal multiple-octet coded character Set (UCS) -- Part 1: Architecture and basic multilingual plane, and its Amendments.

[Normalizer] The Unicode Consortium. Normalization Demo. <http://www.unicode.org/unicode/reports/tr15/Normalizer.html>

[NormTest] Mark Davis. Unicode Normalization Test Suite. <http://www.unicode.org/Public/UNIDATA/NormalizationTest.txt>

[RFC2119] Scott Bradner. Key words for use in RFCs to Indicate Requirement Levels, March 1997. <http://www.ietf.org/rfc/rfc2119.txt>

[RFC 2277] Harald Alvestrand, IETF Policy on Character Sets and Languages, January 1998. <http://www.ietf.org/rfc/rfc2781.txt>

[RFC 2279] Francois Yergeau. UTF-8, a transformation format of ISO 10646. <http://www.ietf.org/rfc/rfc2781.txt>

[RFC 2781] Paul Hoffman and Francois Yergeau. UTF-16, an encoding of ISO 10646. <http://www.ietf.org/rfc/rfc2781.txt>

[Unicode] The Unicode Consortium. The Unicode Standard, Version 3.0. Reading, MA, Addison-Wesley Developers Press, 2000. ISBN 0-201-61633-5.

[UniData] The Unicode Consortium. UnicodeData File. <ftp://ftp.unicode.org/Public/UNIDATA/UnicodeData.txt> For explanation on the content of this file, please see

<ftp://ftp.unicode.org/Public/UNIDATA/UnicodeData.html>

[UniPolicy] The Unicode Consortium. Unicode Consortium Policies. <http://www.unicode.org/unicode/standard/policies>

[UTR15] Mark Davis and Martin Duerst. Unicode Normalization Forms. Unicode Technical Report #15, Version 18.0. <http://www.unicode.org/unicode/reports/tr15/tr15-18.html>, also on the CD of [Unicode].

[windows-1258] Microsoft Windows Codepage: 1258 (Viet Nam). http://www.microsoft.com/globaldev/reference/sbcs/1258.htm>

Copyright

Author's Addresses

Martin J. Duerst
W3C/Keio University
5322 Endo, Fujisawa
252-8520 Japan
mailto:duerst@w3.org
http://www.w3.org/People/D%C3%BCrst/

Tel/Fax: +81 466 49 1170

Note: Please write "Duerst" with u-umlaut wherever
possible, e.g. as "D&252;rst" in HTML and XML.

Mark E. Davis
IBM Center for Java Technology
10275 North De Anza Bouleward
Cupertino 95014 CA
U.S.A.
mailto:mark.davis@us.ibm.com
http://www.macchiato.com
Tel: +1 (408) 777-5850
Fax: +1 (408) 777-5891