

Technische Dokumente - RFC3454



Network Working Group
Request for Comments: 3454
Category: Standards Track

P. Hoffman
IMC & VPNC
M. Blanchet
Viagenie
December 2002

Preparation of Internationalized Strings ("stringprep")

Status of this Memo

This document specifies an Internet standards track protocol for the Internet community, and requests discussion and suggestions for improvements. Please refer to the current edition of the "Internet Official Protocol Standards" (STD 1) for the standardization state and status of this protocol. Distribution of this memo is unlimited.

Copyright Notice

Copyright (C) The Internet Society (2002). All Rights Reserved.

Abstract

This document describes a framework for preparing Unicode text strings in order to increase the likelihood that string input and string comparison work in ways that make sense for typical users throughout the world. The stringprep protocol is useful for protocol identifier values, company and personal names, internationalized domain names, and other text strings.

This document does not specify how protocols should prepare text strings. Protocols must create profiles of stringprep in order to fully specify the processing options.

Table of Contents

1. Introduction.....	3
1.1 Terminology.....	4
1.2 Using stringprep in protocols.....	4
2. Preparation Overview.....	6
3. Mapping.....	7
3.1 Commonly mapped to nothing.....	7
3.2 Case folding.....	8
4. Normalization.....	9
5. Prohibited Output.....	10
5.1 Space characters.....	11
5.2 Control characters.....	11
5.3 Private use.....	12

Dokument RFC3454

RFC 3454 Preparation of Internationalized Strings December 2002

5.4 Non-character code points.....	12
5.5 Surrogate codes.....	13
5.6 Inappropriate for plain text.....	13
5.7 Inappropriate for canonical representation.....	13
5.8 Change display properties or deprecated.....	13
5.9 Tagging characters.....	14
6. Bidirectional Characters.....	14
7. Unassigned Code Points in Stringprep Profiles.....	15
7.1 Categories of code points.....	16
7.2 Reasons for difference between stored strings and queries...17	
7.3 Versions of applications and stored strings.....	18
8. References.....	19
8.1 Normative references.....	19
8.2 Informative references.....	19
9. Security Considerations.....	19
9.1 Stringprep-specific security considerations.....	19
9.2 Generic Unicode security considerations.....	20
10. IANA Considerations.....	21
11. Acknowledgements.....	22
A. Unicode repertoires.....	23
A.1 Unassigned code points in Unicode 3.2.....	23
B. Mapping Tables.....	31
B.1 Commonly mapped to nothing.....	31
B.2 Mapping for case-folding used with NFKC.....	32
B.3 Mapping for case-folding used with no normalization.....	61
C. Prohibition tables.....	78
C.1 Space characters.....	78
C.1.1 ASCII space characters.....	78
C.1.2 Non-ASCII space characters.....	79
C.2 Control characters.....	79
C.2.1 ASCII control characters.....	79
C.2.2 Non-ASCII control characters.....	79
C.3 Private use.....	80
C.4 Non-character code points.....	80
C.5 Surrogate codes.....	80
C.6 Inappropriate for plain text.....	80
C.7 Inappropriate for canonical representation.....	81
C.8 Change display properties or are deprecated.....	81
C.9 Tagging characters.....	81
D. Bidirectional tables.....	81
D.1 Characters with bidirectional property "R" or "AL".....	81
D.2 Characters with bidirectional property "L".....	82
Authors' Addresses.....	90
Full Copyright Statement.....	91

1. Introduction

Application programs can display text in many different ways. Similarly, a user can enter text into an application program in a myriad of fashions. Internationalized text (that is, text that is not restricted to the narrow set of US-ASCII characters) has many input and display behaviors that make it difficult to compare text in a consistent fashion.

This document specifies a framework of processing rules for Unicode text. Other protocols can create profiles of these rules; these profiles will allow users to enter internationalized text strings in applications and have the highest chance of getting the content of the strings correct. In this case, "correct" means that if two different people enter what they think is the same string into two different input mechanisms, the strings should match on a character-by-character basis.

This framework does not describe how data is transcoded from other character sets into Unicode. In systems that uses non-Unicode character sets, the transcoding algorithm is a critical part of enabling secure and "correct" operation of internationalized text strings.

In addition to helping string matching, profiles of stringprep can also exclude characters that should not normally appear in text that is used in the protocol. The profile can prevent such characters by changing the characters to be excluded to other characters, by removing those characters, or by causing an error if the characters would appear in the output. For example, because the backspace character can cause unpredictable display results, a profile can specify that a string containing a backspace character would cause an error.

A profile of stringprep converts a single string of input characters to a string of output characters, or returns an error if the output string would contain a prohibited character. Stringprep profiles cannot both emit a string and return an error.

Stringprep profiles cannot account for all of the variations that might occur or that a user might expect. In particular, a profile will not be able to account for choice of spellings in all languages for all scripts because the number of alternative spellings of words and phrases is immense. Users would probably expect all spelling equivalents to be made equivalent, or none of them to be. Examples of spelling equivalents include "theater" vs. "theatre", and "hemoglobin" vs. "hmoglobin" in American vs. British English. Other examples are simplified Chinese spellings of names (for

example,"") vs. the equivalent traditional Chinese spelling (for example, ""). Language-specific equivalences such as "Aepfel" vs. "pfel", which are sometimes considered equivalent in German, may not be considered equivalent in other languages.

1.1 Terminology

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14, RFC 2119 [RFC2119].

Note: A glossary of terms used in Unicode and ISO/IEC 10646 can be found in [Glossary]. Information on the 10646/Unicode character encoding model can be found in [CharModel].

Character names in this document use the notation for code points and names from the Unicode Standard [Unicode3.2] and ISO/IEC 10646 [ISO10646]. For example, the letter "a" may be represented as either "U+0061" or "LATIN SMALL LETTER A". In the lists of mappings and the prohibited characters, the "U+" is left off to make the lists easier to read. The comments for character ranges are shown in square brackets (such as "[CONTROL CHARACTERS]") and do not come from the standards.

1.2 Using stringprep in protocols

The stringprep protocol does not stand on its own; it has to be used by other protocols at precisely-defined places in those other protocols. For example, a protocol that has strings that come from the entire ISO/IEC 10646 [ISO10646] character repertoire might specify that only strings that have been processed with a particular profile of stringprep are legal. Another example would be a protocol that does string comparison as a step in the protocol; that protocol might specify that such comparison is done only after processing the strings with a specific profile of stringprep.

When two protocols that use different profiles of stringprep interoperate, there may be conflict about what characters are and are not allowed in the final string. Thus, protocol developers should strongly consider re-using existing profiles of stringprep.

When developers wish to allow users as wide of a range of characters as possible in input text strings, they should, where possible, cause stringprep to convert characters from the input string to a canonical form instead of prohibiting them.

Dokument RFC3454

RFC 3454 Preparation of Internationalized Strings December 2002

Although it would be easy to use the stringprep process to "correct" perceived mis-features or bugs in the current character standards, stringprep profiles SHOULD NOT do so.

A profile of stringprep can create tables different from those in the appendixes of this document, but it will be an exception when they do. The intention of stringprep is to define the tables and have the profiles of stringprep select among those defined tables.

A profile of stringprep MUST include all of the following:

- The intended applicability of the profile
- The character repertoire that is the input and output to stringprep (which is Unicode 3.2 for this version of stringprep)
- The mapping tables from this document used (as described in section 3)
- Any additional mapping tables specific to the profile
- The Unicode normalization used, if any (as described in section 4)
- The tables from this document of characters that are prohibited as output (as described in section 5)
- The bidirectional string testing used, if any (as described in section 6)
- Any additional characters that are prohibited as output specific to the profile

Each profile MUST state the character repertoire on which the profile will operate. Appendix A lists the Unicode repertoires that can be selected. No repertoire is ever complete, and it is expected that characters will be added to the Unicode repertoire for the foreseeable future. Section 7 of this document describes how to handle characters that are assigned in later versions of the Unicode repertoires. Subsections of appendix A also list unassigned code points for each repertoire.

This document is for Unicode version 3.2, and should not be considered to automatically apply to later Unicode versions. The IETF, through an explicit standards action, may update this document as appropriate to handle later Unicode versions.

This document lists the unassigned code points in the range 0 to 10FFFF for Unicode 3.2 in appendix A. The list in appendix A MUST be used by implementations of this specification. If there are any discrepancies between the list in appendix A and the Unicode 3.2 specification, the list in appendix A always takes precedence.

Each profile of stringprep MUST be registered with IANA. The registration procedure is described in the IANA Considerations appendix; basically, the IESG must review each profile of stringprep. Protocol developers are strongly encouraged to look through the IANA profile registry when creating new profiles for stringprep, and to re-use logic from earlier profiles where possible in new profiles. In some cases, an existing profile can be reused by a different protocol.

2. Preparation Overview

The steps for preparing strings are:

- 1) Map -- For each character in the input, check if it has a mapping and, if so, replace it with its mapping. This is described in section 3.
- 2) Normalize -- Possibly normalize the result of step 1 using Unicode normalization. This is described in section 4.
- 3) Prohibit -- Check for any characters that are not allowed in the output. If any are found, return an error. This is described in section 5.
- 4) Check bidi -- Possibly check for right-to-left characters, and if any are found, make sure that the whole string satisfies the requirements for bidirectional strings. If the string does not satisfy the requirements for bidirectional strings, return an error. This is described in section 6.

The above steps MUST be performed in the order given to comply with this specification.

The mappings described in section 3, and the optional Unicode normalization described in section 4, can be one-to-none, one-to-one, one-to-many, many-to-one, or many-to-many. That is, some characters might be eliminated or replaced by more than one character, and the output of this step might be shorter or longer than the input. Because of this, the system using stringprep MUST be prepared to receive a longer or shorter string than the one input in the stringprep algorithm.

3. Mapping

Each character in the input stream MUST be checked against a mapping table. The mapping table SHOULD come from this document, although the mapping table MAY be added to or altered by the profile. The mapping tables are subsections of appendix B.

The lists in appendix B MUST be used by implementations of this specification. If there are any discrepancies between the lists in appendix B and subsections below, the lists in appendix B always takes precedence.

For any individual character, the mapping table MAY specify that a character be mapped to nothing, or mapped to one other character, or mapped to a string of other characters.

Mapped characters are not re-scanned during the mapping step. That is, if character A at position X is mapped to character B, character B which is now at position X is not checked against the mapping table.

3.1 Commonly mapped to nothing

The following characters are simply deleted from the input (that is, they are mapped to nothing) because their presence or absence in protocol identifiers should not make two strings different. They are listed in Table B.1.

Some characters are only useful in line-based text, and are otherwise invisible and ignored.

00AD; SOFT HYPHEN
1806; MONGOLIAN TODO SOFT HYPHEN
200B; ZERO WIDTH SPACE
2060; WORD JOINER
FEFF; ZERO WIDTH NO-BREAK SPACE

Some characters affect glyph choice and glyph placement, but do not bear semantics.

034F; COMBINING GRAPHEME JOINER
180B; MONGOLIAN FREE VARIATION SELECTOR ONE
180C; MONGOLIAN FREE VARIATION SELECTOR TWO
180D; MONGOLIAN FREE VARIATION SELECTOR THREE
200C; ZERO WIDTH NON-JOINER
200D; ZERO WIDTH JOINER
FE00; VARIATION SELECTOR-1
FE01; VARIATION SELECTOR-2

FE02; VARIATION SELECTOR-3
FE03; VARIATION SELECTOR-4
FE04; VARIATION SELECTOR-5
FE05; VARIATION SELECTOR-6
FE06; VARIATION SELECTOR-7
FE07; VARIATION SELECTOR-8
FE08; VARIATION SELECTOR-9
FE09; VARIATION SELECTOR-10
FE0A; VARIATION SELECTOR-11
FE0B; VARIATION SELECTOR-12
FE0C; VARIATION SELECTOR-13
FE0D; VARIATION SELECTOR-14
FE0E; VARIATION SELECTOR-15
FE0F; VARIATION SELECTOR-16

3.2 Case folding

If a profile is going to map characters for case-insensitive comparison, that profile SHOULD map using either appendix B.2 or appendix B.3. appendix B.2 is for profiles that also use Unicode normalization form KC, while appendix B.3 is for profiles that do not use Unicode normalization. These tables map from uppercase to lowercase characters. Note that this could have been "change all lowercase characters into uppercase characters". However, the upper-to-lower folding was chosen because there is a tradition of using lowercase in current Internet applications and protocols.

If a profile creates its own mapping tables for case folding, they SHOULD be based on [UTR21], and SHOULD map from uppercase characters to lowercase. The "CaseFolding.txt" file from the Unicode database SHOULD be used to prepare the mapping table. The profile SHOULD do full case mapping (that is, using statuses C, F, and I).

If the profile is using Unicode normalization form KC (as described in section 4 of this document), it is important to note that there are some characters that do not have mappings in [UTR21] but still need processing. These characters include a few Greek characters and many symbols that contain Latin characters. The list of characters to add to the mapping table can be determined by the following algorithm:

```
b = NormalizeWithKC(Fold(a));  
c = NormalizeWithKC(Fold(b));  
if c is not the same as b, add a mapping for "a to c".
```

Because NormalizeWithKC(Fold(c)) always equals c, the table is stable from that point on.

Appendix B.3 is derived from the CaseFolding-3.txt file associated with Unicode 3.2; appendix B.2 is based on appendix B.3 with the additional characters added from the algorithm above.

Authors of profiles of this document need to consider the effects of changing the mapping of any currently-assigned character when updating their profiles. Adding a new mapping for a currently-assigned character, or changing an existing mapping, could cause a variance between the behavior of systems that have been updated and systems that have not been updated.

4. Normalization

The output of the mapping step is optionally normalized using one of the Unicode normalization forms, as described in [UAX15]. A profile can specify one of two options for Unicode normalization:

- no normalization
- Unicode normalization with form KC

A profile MAY choose to do no normalization. However, such a profile can easily yield results that will be surprising to typical users, depending on the input mechanism they use. For example, some input mechanisms enter compatibility characters that look exactly like the underlying characters, but have different code points. Another example of where Unicode normalization helps create predictable results is with characters that have multiple combining diacritics: normalization orders those diacritics in a predictable fashion.

On the other hand, Unicode normalization requires fairly large tables and somewhat complicated character reordering logic. The size and complexity should not be considered daunting except in the most restricted of environments, and needs to be weighed against the problems of user surprise from comparing unnormalized strings. Note that the tables used for normalization are not given in this document, but instead must be derived from the Unicode database, as described in [UAX15].

There is a third form of normalization, Unicode normalization with form C. If a profile is going to use a Unicode normalization, it MUST use Unicode normalization form KC. Form KC maps many "compatibility characters" to their equivalents. Some user interface systems make it possible to enter compatibility characters instead of the base equivalents. Thus, using form KC instead of form C will cause more strings that users would expect to match to actually match.

A profile that specifies Unicode normalization MUST use the normalization in [UAX15] that is associated with the version of the Unicode character set specified for the profile.

The composition process described in [UAX15] requires a fixed composition version of Unicode to ensure that strings normalized under one version of Unicode remain normalized under all future versions of Unicode.

The IETF is relying on Unicode not to change the normalization of currently-assigned characters in future versions of normalization. If a future version of the normalization tables changes the normalized value of an existing character, authors of profiles of this document have to look at the changes very carefully before they update their normalization tables. Such a change could cause a variance between the behavior of systems that have been updated and systems that have not been updated.

5. Prohibited Output

Before the text can be emitted, it MUST be checked for prohibited code points. There are a variety of prohibited code points, as described in this section. A profile of this document MAY use all or some of the tables in appendix C.

The stringprep process never emits both an error and a string. If an error is detected during the checking for prohibited code points, only an error is returned.

Note that the subsections below describe how the tables in appendix C were formed. They are here for people who want to understand more, but they should be ignored by implementors. Implementations that use tables MUST map based on the tables themselves, not based on the descriptions in this section of how the tables were created.

The lists in appendix C MUST be used by implementations of this specification. If there are any discrepancies between the lists in appendix C and subsections below, the lists in appendix C always take precedence.

Some code points listed in one section may also appear in other sections.

It is important to note that a profile of this document MAY prohibit additional characters.

Each subsection of this section has a matching subsection in appendix C. For example, the characters listed in section 5.1 are listed in appendix C.1.

5.1 Space characters

Space characters can make accurate visual transcription of strings nearly impossible and could lead to user entry errors in many ways. Note that the list below is split into two tables in appendix C: Table C.1.1 contains the ASCII code points, while Table C.1.2 contains the non-ASCII code points. Most profiles of this document that want to prohibit space characters will want to include both tables.

0020; SPACE
00A0; NO-BREAK SPACE
1680; OGHAM SPACE MARK
2000; EN QUAD
2001; EM QUAD
2002; EN SPACE
2003; EM SPACE
2004; THREE-PER-EM SPACE
2005; FOUR-PER-EM SPACE
2006; SIX-PER-EM SPACE
2007; FIGURE SPACE
2008; PUNCTUATION SPACE
2009; THIN SPACE
200A; HAIR SPACE
200B; ZERO WIDTH SPACE
202F; NARROW NO-BREAK SPACE
205F; MEDIUM MATHEMATICAL SPACE
3000; IDEOGRAPHIC SPACE

5.2 Control characters

Control characters (or characters with control function) cannot be seen and can cause unpredictable results when displayed. Note that the list below is split into two tables in appendix C: Table C.2.1 contains the ASCII code points, while Table C.2.2 contains the non-ASCII code points. Most profiles of this document that want to prohibit control characters will want to include both tables.

0000-001F; [CONTROL CHARACTERS]
007F; DELETE
0080-009F; [CONTROL CHARACTERS]
06DD; ARABIC END OF AYAH
070F; SYRIAC ABBREVIATION MARK
180E; MONGOLIAN VOWEL SEPARATOR

200C; ZERO WIDTH NON-JOINER
 200D; ZERO WIDTH JOINER
 2028; LINE SEPARATOR
 2029; PARAGRAPH SEPARATOR
 2060; WORD JOINER
 2061; FUNCTION APPLICATION
 2062; INVISIBLE TIMES
 2063; INVISIBLE SEPARATOR
 206A-206F; [CONTROL CHARACTERS]
 FEFF; ZERO WIDTH NO-BREAK SPACE
 FFF9-FFFC; [CONTROL CHARACTERS]
 1D173-1D17A; [MUSICAL CONTROL CHARACTERS]

5.3 Private use

Because private-use characters do not have defined meanings, they are likely to be prohibited. The private-use characters are:

E000-F8FF; [PRIVATE USE, PLANE 0]
 F0000-FFFFD; [PRIVATE USE, PLANE 15]
 100000-10FFFFD; [PRIVATE USE, PLANE 16]

5.4 Non-character code points

Non-character code points are code points that have been allocated in ISO/IEC 10646 but are not characters. Because they are already assigned, they are guaranteed not to later change into characters.

FDD0-FDEF; [NONCHARACTER CODE POINTS]
 FFFE-FFFF; [NONCHARACTER CODE POINTS]
 1FFFE-1FFFF; [NONCHARACTER CODE POINTS]
 2FFFE-2FFFF; [NONCHARACTER CODE POINTS]
 3FFFE-3FFFF; [NONCHARACTER CODE POINTS]
 4FFFE-4FFFF; [NONCHARACTER CODE POINTS]
 5FFFE-5FFFF; [NONCHARACTER CODE POINTS]
 6FFFE-6FFFF; [NONCHARACTER CODE POINTS]
 7FFFE-7FFFF; [NONCHARACTER CODE POINTS]
 8FFFE-8FFFF; [NONCHARACTER CODE POINTS]
 9FFFE-9FFFF; [NONCHARACTER CODE POINTS]
 AFFFE-AFFFF; [NONCHARACTER CODE POINTS]
 BFFFE-BFFFF; [NONCHARACTER CODE POINTS]
 CFFFE-CFFFF; [NONCHARACTER CODE POINTS]
 DFFFE-DFFFF; [NONCHARACTER CODE POINTS]
 EFFFE-EFFFF; [NONCHARACTER CODE POINTS]
 FFFFE-FFFFF; [NONCHARACTER CODE POINTS]
 10FFFE-10FFFF; [NONCHARACTER CODE POINTS]

The non-character code points are listed in the PropList.txt file from the Unicode database.

5.5 Surrogate codes

The following code points are permanently reserved for use as surrogate code values in the UTF-16 encoding, will never be assigned to characters in the Unicode repertoire, and are therefore prohibited:

D800-DFFF; [SURROGATE CODES]

5.6 Inappropriate for plain text

The following characters do not appear in regular text.

FFF9; INTERLINEAR ANNOTATION ANCHOR
FFFA; INTERLINEAR ANNOTATION SEPARATOR
FFFB; INTERLINEAR ANNOTATION TERMINATOR
FFFC; OBJECT REPLACEMENT CHARACTER

Although the replacement character (U+FFFD) might be used when a string is displayed, it doesn't make sense for it to be part of the string itself. It is often displayed by renderers to indicate "there would be some character here, but it cannot be rendered". For example, on a computer with no Asian fonts, a string with three ideographs might be rendered with three replacement characters.

FFFD; REPLACEMENT CHARACTER

5.7 Inappropriate for canonical representation

The ideographic description characters allow different sequences of characters to be rendered the same way, which makes them inappropriate for strings that have to have a single canonical representation.

2FF0-2FFB; [IDEOGRAPHIC DESCRIPTION CHARACTERS]

5.8 Change display properties or are deprecated

The following characters can cause changes in display or the order in which characters appear when rendered, or are deprecated in Unicode.

0340; COMBINING GRAVE TONE MARK
0341; COMBINING ACUTE TONE MARK
200E; LEFT-TO-RIGHT MARK
200F; RIGHT-TO-LEFT MARK

202A; LEFT-TO-RIGHT EMBEDDING
 202B; RIGHT-TO-LEFT EMBEDDING
 202C; POP DIRECTIONAL FORMATTING
 202D; LEFT-TO-RIGHT OVERRIDE
 202E; RIGHT-TO-LEFT OVERRIDE
 206A; INHIBIT SYMMETRIC SWAPPING
 206B; ACTIVATE SYMMETRIC SWAPPING
 206C; INHIBIT ARABIC FORM SHAPING
 206D; ACTIVATE ARABIC FORM SHAPING
 206E; NATIONAL DIGIT SHAPES
 206F; NOMINAL DIGIT SHAPES

5.9 Tagging characters

The following characters are used for tagging text and are invisible.

E0001; LANGUAGE TAG
 E0020-E007F; [TAGGING CHARACTERS]

6. Bidirectional Characters

Most characters are displayed from left to right, but some are displayed from right to left. This feature of Unicode is called "bidirectional text", or "bidi" for short. The Unicode standard has an extensive discussion of how to reorder glyphs for display when dealing with bidirectional text such as Arabic or Hebrew. See [UAX9] for more information. In particular, all Unicode text is stored in logical order.

A profile MAY choose to ignore bidirectional text. However, ignoring bidirectional text can cause display ambiguities. For example, it is quite easy to create two different strings with the same characters (but in different order) that are correctly displayed identically. Therefore, in order to avoid most problems with ambiguous bidirectional text display, profile creators should strongly consider including the bidirectional character handling described in this section in their profile.

The stringprep process never emits both an error and a string. If an error is detected during the checking of bidirectional strings, only an error is returned.

[Unicode3.2] defines several bidirectional categories; each character has one bidirectional category assigned to it. For the purposes of the requirements below, an "RandALCat character" is a character that has Unicode bidirectional categories "R" or "AL"; an "LCat character" is a character that has Unicode bidirectional category "L". Note

that there are many characters which fall in neither of the above definitions; Latin digits (through) are examples of this because they have bidirectional category "EN".

In any profile that specifies bidirectional character handling, all three of the following requirements MUST be met:

- 1) The characters in section 5.8 MUST be prohibited.
- 2) If a string contains any RandALCat character, the string MUST NOT contain any LCat character.
- 3) If a string contains any RandALCat character, a RandALCat character MUST be the first character of the string, and a RandALCat character MUST be the last character of the string.

Note that requirement 3 prohibits strings such as ("aleph 1") but allows strings such as ("aleph 1 beh"). [UAX9] goes into great detail about the display order of strings that contain particular categories of characters in particular sequences.

Table D.1 lists the characters that belong to Unicode bidirectional categories "R" and "AL". Table D.2 lists all the characters that belong to Unicode bidirectional category "L". These tables are derived from [Unicode3.2].

7. Unassigned Code Points in Stringprep Profiles

This section describes two different types of strings in typical protocols where internationalized strings are used: "stored strings" and "queries". Of course, different Internet protocols use strings very differently, so these terms cannot be used exactly in every protocol that needs to use stringprep. In general, "stored strings" are strings that are used in protocol identifiers and named entities, such as names in digital certificates and DNS domain name parts. "Queries" are strings that are used to match against strings that are stored identifiers, such as user-entered names for digital certificate authorities and DNS lookups.

All code points not assigned in the character repertoire named in a stringprep profile are called "unassigned code points". Stored strings using the profile MUST NOT contain any unassigned code points. Queries for matching strings MAY contain unassigned code points. Note that this is the only part of this document where the requirements for queries differs from the requirements for stored strings.

Using two different policies for where unassigned code points can appear removes the need for versioning in protocols that use stringprep profiles. This is very useful since it makes the overall processing simpler and does not impose a "protocol" to handle versioning. It is expected that the ISO/IEC 10646 and Unicode repertoires will be updated fairly frequently; at the time that this document is being written, it has happened approximately once a year. Each time a new version of a repertoire appears, a new version of a profile MAY be created. Some end users will want to use the new code points as soon as they are defined.

The list of unassigned code points MUST be given in a profile, and that list MUST be used by implementations of the profile.

The goal of the requirements in this section is to prevent comparisons between two strings that were both permitted to contain unassigned code points. When two strings X and Y are compared and string Y was prepared in a way that permits unassigned code points, a negative result to the comparison is not definitive; it's possible that the strings don't match even though they would match if a more recent version of the profile were used for Y. However, if both X and Y were prepared in a way that permits unassigned code points, something worse can happen: even a positive result for the comparison is not definitive. It is possible that the strings do match even though they would not match if a more recent version of the profile were used (one that prohibits a code point appearing in both X and Y).

Due to the way that versioning is handled in this section, stored strings that are embedded in structures that cannot be changed (such as the signed parts of digital certificates) MUST NOT contain any unassigned code points.

7.1 Categories of code points

Each code point in a repertoire named by a profile of stringprep can be categorized by how it acts in the process described in earlier sections of this document:

- AO Code points that can be in the output
- MN Code points that cannot be in the output because they never appear as output from mapping or normalization
- D Code points that cannot be in the output because they are disallowed in the prohibition step
- U Unassigned code points

A subsequent version of a profile that references a newer version of a repertoire with new code points will inherently have some code points move from category U to either D, MN, or AO. For backwards compatibility, a subsequent version of a profile MUST NOT move code points from any other category. That is, current AO, MN, or D code points MUST NOT ever change to a different category.

Stored strings MUST NOT contain any code points outside of AO for the latest version of a profile. That is, they are forbidden to contain code points from the MN, D, or U categories.

Applications creating queries MUST treat U code points as if they were AO when preparing the query to be entered in the process described by a profile of stringprep. Those applications MAY optionally have a preprocessor that provide stricter checks: treating unassigned code points in the input as errors, or warning the user about the fact that the code point is unassigned in the version of a profile that the software is based on; such a choice is a local matter for the software.

7.2 Reasons for the difference between stored strings and queries

Different software using different versions of a stringprep profile need to interoperate with maximal compatibility. The scheme described in this section (stored strings MUST NOT contain unassigned code points, queries MAY include unassigned code points) allows that compatibility without introducing any known security or interoperability issues.

The list below shows what happens if a query contains a code point from category U that is allowed in a newer version of a profile. The query either matches the string that was intended, or matches no string at all. In this list, the query comes from an application using version "oldVersion" of a profile, the stored string was created using version "newVersion" of the same profile, and the code point X was in category U in oldVersion, and has changed category to AO, MN, or D. There are 3 possible scenarios:

1. X is assigned to AO -- In newVersion, X is in category AO. Because the application passed X through, it gets back a positive match with the stored string. There is one exceptional case, where X is a combining mark.

The order of combining marks is normalized, so if another combining mark Y has a lower combining class than X then XY will be put in the canonical order YX. (Unassigned code points are never reordered, so this doesn't happen in oldVersion). If the query contains YX, the query will get positive match with the

stored string. However, no string can be stored with XY, so a query with XY will get a negative answer to the test for matching.

2. X is assigned to MN -- In newVersion, X is normalized to code point "nX" and therefore X is now put in category MN. This cannot exist in any stored string, so any query containing X will get a negative answer to the test for matching. Note, however, if the query had contained the letter nX, it would have positively matched.
3. X is assigned to D -- In newVersion, X is in category D. This cannot exist in any stored string, so any query containing X will get a negative answer to the test for matching.

In none of the cases does the query get data for a stored string other than the one it actually tried to match against.

Profiles are stable between versions in the following sense: If a string S has been prepared using newVersion, then it will not change if it is subsequently prepared using oldVersion.

7.3 Versions of applications and stored strings

Another way to see that this versioning system works is to compare what happens when an application uses a newer or older version of a profile.

Newer query application -- Suppose that a querying application is using version newVersion and the stored string was created using version oldVersion. This case is simple: there will be no characters in the stored string that cannot be queried by the application because the new profile uses a superset of the code points used for making the stored string.

Newer stored string -- Suppose that a querying application is using oldVersion and the stored string was created using a profile that uses newVersion. Because the querying application let unassigned code points pass through, the user can query on stored strings that use code points in newVersion. No stored strings can have code points that are unassigned in newVersion, since that is illegal. In order to get a match, the querying application has to enter the unassigned code points in the proper order, and has to use unassigned code points that would make it through both the mapping and the normalization steps.

Dokument RFC3454

RFC 3454 Preparation of Internationalized Strings December 2002

8. References

8.1 Normative references

[UAX15] Mark Davis and Martin Duerst. Unicode Standard Annex
#15: Unicode Normalization Forms, Version 3.2.0.