

Title:	Proposed additions to Principles and Procedures document
Source:	US and Unicode (drafted by V.S. Umamaheswaran, umavs@ca.ibm.com)
Reference:	Principles and Procedures document - N2902
Action:	For consideration and acceptance at WG2 meeting M47
Distribution:	ISO/IEC JTC 1/SC 2/WG 2

(Note: The section numbers below are referencing sections in document N2902 -- see www.dkuug.dk/jtc1/sc2/wg2/docs/n2902.doc)

1. Add a new section F.5

F.5 Some additional guiding principles

An analysis of the following three additions to Amendment 1 to 10646: 2003 has shown some particular difficulties for existing implementations (see document N2987).

- a. Addition of HEBREW POINT QAMATS QATAN per resolution M45.4 item j to distinguish it from use of HEBREW POINT QAMATS as GADOL in some orthographies that distinguish these. Here an annotation was made to the existing character.
- b. Addition of HEBREW ACCENT ATNAH HAFUKH per resolution M45.4 item k to distinguish it from use of HEBREW ACCENT YERAH BEN YOMO as GALGAL in some orthographies that distinguish these. Here the glyph for the existing character was given to the new character, changing the glyph of the existing character to be more aligned with the character names.
- c. Addition of LATIN CAPITAL LETTER GLOTTAL STOP to cater for orthographies that use the phonetic symbol LATIN LETTER GLOTTAL STOP as a lower case letter (per resolution M45.5 item a).

Based on this analysis of these cases of disunification, to preserve the pre-disunification use of existing characters also after disunification, some additional guiding principles are provided here.

F5.1 The representative glyphs of existing characters will not be changed in such a way as to change their identity, and the range of glyphs expected for existing characters will not increase as a result of disunification.

F5.2 Very significant character properties (such as case) for existing characters shall not be changed, because of the large risk of adverse impact on existing implementations of the standard.

If a character disunification cannot be achieved by adding one new character without requiring a change in very significant properties of the existing character and without changing the representative glyph or range of expected glyphs for the existing character, then new characters will be added for each of the distinct, specific letterforms required. The existing character will not be intended for use in scenarios in which the distinct, specific letterforms are used. This may result in visually duplicate characters, which may be necessary under the above conditions. While it is desirable that a character name be fully appropriate to the given character and its representative glyph, concern over less-than-ideal names will not provide a sufficient basis for overriding these guidelines. Exceptions will be permitted only after careful consideration of hits on existing implementations and on the basis of substantial rationale.

Rationale:

An analysis of three new characters added to Amendment 1 to ISO/IEC 10646: 2003, disunified from existing characters showed some implementation difficulties that could have been avoided. See attachment 1 for details. The above guiding principles are proposed to avoid future hits on implementation of existing characters.

2. Add new section D.2.5 and a pointer from A.1 Submitter's responsibilities if any of the proposed characters can be Syntax characters.

(Rationale: The wording in the proposed text below is self-explanatory).

D.2.5 Reserved code points for stability of identifiers

Implementers of programming languages, markup languages, scripting languages, regular expression engines, character-based protocols, and similar programs or systems require the ability to clearly distinguish between characters that can serve in identifiers, and those that are for syntactic elements. Moreover, a high degree of stability is required. To provide the necessary level of stability, all of the reserved code points in the following blocks are reserved for syntax characters.

[U+2300-U+23FF]	Miscellaneous_Technical
[U+2400-U+243F]	Control_Pictures
[U+2440-U+245F]	Optical_Character_Recognition
[U+2600-U+26FF]	Miscellaneous_Symbols
[U+2700-U+27BF]	Dingbats
[U+27C0-U+27EF]	Miscellaneous_Mathematical_Symbols_A
[U+2B00-U+2BFF]	Miscellaneous_Symbols_And_Arrows
[U+2E00-U+2E7F]	Supplemental_Punctuation

What this means is that **no** new letters suitable for identifiers (letters, combining marks, or numbers) will be allocated in these ranges. In addition, it is strongly encouraged (but not required) that any new characters that are suitable as programmatic syntax characters be allocated in these blocks. (For more information, see *Unicode Standard Annex #31 Identifier and Pattern Syntax* at <http://www.unicode.org/reports/tr31/>.) "

Attachment 1 --L2/05-057 - Proposed Principles for Character Disunifications, Peter Constable, Microsoft, 2005-2-2

Attachment 2 - Unicode Standard Annex #31, Identifier and Pattern Syntax, Version 4.1.0, 2005-03-25

Proposed Principles for Character Disunifications

Peter Constable, Microsoft
2005-2-2

In the past year, UTC has approved various character disunifications – encoding new characters to create distinctions that were not previously made.¹ For implementers, these have been a mixed bag: some present no significant problems; others, however, were done in ways that leave implementers facing some significant problems that could have been avoided. To avoid such problems in the future, I propose that UTC adopt certain principles that guide how character disunifications should be handled.

Three particular disunifications are considered here: QAMATS, YERAH BEN YOMO and GLOTTAL STOP. I will describe each, explaining why the disunification of YERAH BEN YOMO and GLOTTAL STOP have resulted in problems while the disunification of QAMATS does not. By considering these three cases, some simple principles can be identified that can serve to avoid similar problems in the future.

Disunification of QAMATS

The Hebrew mark *qamats* is one of the vowel points used in pointed Hebrew text. While historically there was only one mark, it can be used to write two different vowel pronunciations. This led in recent times to publishers creating a glyph distinction in order to distinguish the two readings.

Most users do not make this distinction in texts; for them, the existing character U+05B8 HEBREW POINT QAMATS has been adequate. For those that wish to make the distinction, however, two characters are needed: *qamats gadol*, and a separate character *qamats qatan*. The latter typically differs from the former in having a longer stem.



Figure 1. Contrast between *qamats gadol* (short stem) and *qamats qatan* (long stem)

What was proposed and accepted by UTC was to leave the existing character U+05B8 HEBREW POINT QAMATS as it is, and to encode a new character U+05BA HEBREW POINT QAMATS QATAN. Because the existing character was not changed, existing implementations are unaffected, and users that do not make the distinction can continue to use it, regardless of whether the

¹ This discussion applies only to disunification of individual characters, not the disunification of entire scripts, such as the decision to encode Coptic separately from Greek.

implementation also supports the new character or not. For users that *do* make the distinction, they use the existing character, though now in fewer instances and with a more restrictive meaning.

Disunification of YERAH BEN YOMO

The Hebrew mark *yerah ben yomo* is one of the accents from the Tiberian accentual system used by Masoretic scribes to indicate textual structure within verses and to provide guidance on the correct chanting of the text. Historically, two similarly-shaped but distinct accents were used, but the distinction was at some point lost. The distinction has been rediscovered in recent years, however, and some users now want to make the distinction in encoded texts.

The existing character U+05AA HEBREW ACCENT YERAH BEN YOMO was encoded without awareness of the distinction, and it has been used in contexts where the distinction is not made. Most users do not make the distinction in texts; for them, this existing character has, thus far, been adequate. Typically, the preferred glyph for users that do not make the distinction is roughly the shape of a small v optionally with a slight vertical stem at the bottom, though the name apparently means “day-old moon”, suggesting a crescent shape.



Figure 2. Glyphs for U+05AA YERAH BEN YOMO from three existing fonts



Figure 3. Nu. 35:5:5 (right) and Ps. 1:3:3 (left) from Snaitch's edition: no contrast between historically-distinct accents (*galgal*—blue highlight—and *atnah hafukh*—red highlight)

For users that *do* wish to make the distinction, two characters are needed: *galgal*, which has roughly a crescent or semi-circular shape, and *atnah hafukh*, which roughly has the shape of a small v with a slight vertical stem.²

² I have refrained from using the name *yerah ben yomo* when describing the situation in which two accents are distinguish, using an alternate name, to avoid any predispositions about which of the two distinct accents might be represented using the existing character.



Figure 4. Nu. 35:5:5 (right) and Ps. 1:3:3 (left) from *Biblia Hebraica Leningradensia*: *galgal* (blue highlight) and *atnah hafukh* (red highlight) are distinguished

The shape of one of the two distinct accents, *atnah hafukh*, matches the representative glyph of the existing character, YERAH BEN YOMO: the small-v shape with a vertical stem. Thus, one might expect that the existing character would be used for *atnah hafukh*, and that a new character, GALGAL, would be added, having a semi-circular shape. This is not what was proposed, however: because the name *yerah ben yomo* suggests a crescent shape, the proposers apparently felt that it would be inaccurate to have a character with that name but a small-v shape while another character was added with the crescent shape.

Thus, what was proposed, and what was accepted by UTC, was to change the representative glyph for the existing character U+05AA HEBREW ACCENT YERAH BEN YOMO to a semi-circular shape, and to encode a new character U+05A2 HEBREW ACCENT ATNAH HAFUKH with the small-v shape. For users that do make the distinction, YERAH BEN YOMO must have a semi-circular shape, but for users that *do not* make the distinction, a small-v shape is required.

Disunification of LATIN LETTER GLOTTAL STOP

The character U+0294 LATIN LETTER GLOTTAL STOP was encoded to represent the phonetic symbol *glottal stop* used in linguistic transcription. In phonetic usage, the character is drawn with a cap-height glyph, but no case distinction is made. At the time it was encoded, there was no usage known that involved a case distinction. Thus, the character name does not include “small” or “capital” as would be used for cased letters. For some reason, though, this character was assigned the general-category property *lowercase letter* (Ll) rather than *letter – other* (Lo).

'ruaħ hattsa'fon, vehaf'ʃemeʃ, hitvake'hu bene't
jo'ter. game'ru, ki ʔet hannitsa'ħon, jin'ħal, 'mi ʃ
ʃo'ver ʔoraħ ʔet bega'dav. pa'taħ 'ruaħ hattsa'fon v

Figure 5. *Glottal stop* in phonetic transcription: cap-height glyph used (IPA 1999, p. 98)

Certain languages with Latin-based orthographies do use glottal stop as a casing character, with an uppercase and lowercase pair. In these orthographies, the capital letter is displayed with a cap-height glyph, while the small letter is displayed with a glyph of roughly x-height.

Chĩa tɬ'i k'e dawheda ts'ɿ̥ nəhdɔ hɔt'e.
 ʔasɿ wɿzɿ whenehtà nɿ le.

Figure 6. Bi-cameral *glottal stops* in orthographic use: lowercase (red highlight) is x-height, uppercase (blue highlight) is cap-height (from Koyina 1983)

Because the existing character has a cap-height glyph, which is what is required for phonetic transcription, it was originally proposed to change the case property of the existing character to *uppercase* and to add a new *lowercase* letter SMALL GLOTTAL STOP with an x-height glyph. Concerns were raised, however, regarding potential problems for existing implementations if the case of the existing character were changed. (E.g. it could affect indexes, file systems or other protocols that use case mapping.)

Therefore, the proposal was changed to leave the existing character as is with its originally-intended usage for phonetic transcription, and to encode *two* new characters, a casing pair, for orthographic usage. The decision of UTC, however, was to leave the existing character as is, but to encode only one new character, U+0241 LATIN CAPITAL LETTER GLOTTAL STOP.

With this UTC decision, those that want to use the existing character U+0294 LATIN LETTER GLOTTAL STOP for phonetic transcription require a cap-height glyph, which is what would be found in existing font implementations. Those that want to use the pair of characters for orthographic purposes, however, require a font that has an x-height glyph for the existing character.

Comparison of the disunifications

The three disunifications described above differ in terms of the ease with which they can be implemented: the *qamats* disunification presents no problems, while the other two disunifications present significant dilemmas for implementers. The reason for the difference is that the disunification of *qamats* left the existing character completely unchanged, while the other two disunifications did not.

The representative glyphs for the characters in question before and after the disunifications are shown in Table 1:

Character	TUS 4.0	TUS 4.1
U+05B8 HEBREW POINT QAMATS		
U+05BA HEBREW POINT QAMATS QATAN	N/A	
U+05AA HEBREW ACCENT YERAH BEN YOMO		
U+05A2 HEBREW ACCENT ATNAH HAFUKH	N/A	
U+0294 LATIN LETTER GLOTTAL STOP		
U+0241 LATIN CAPITAL LETTER GLOTTAL STOP	N/A	

Table 1. Representative glyphs in TUS 4.0 and TUS 4.1

It should be noted that the glyph for LATIN CAPITAL LETTER GLOTTAL STOP does not actually correspond to what is, in fact, used. Rather, it is an invention, created specifically to provide a capital-like contrast to the representative glyph for the existing lowercase letter.

A better comparison can be seen by considering what glyphs are required in different usage contexts: by users that do not require a two-way distinction, and by users that do. This is shown in Table 2:

Character	No distinction required	Two-way distinction required	Note
U+05B8 HEBREW POINT QAMATS			
U+05BA HEBREW POINT QAMATS QATAN	N/A		
U+05AA HEBREW ACCENT YERAH BEN YOMO			
U+05A2 HEBREW ACCENT ATNAH HAFUKH	N/A		
U+0294 LATIN LETTER GLOTTAL STOP			Cap-height glyph required for phonetic transcription; x-height glyph required for orthographic usage.
U+0241 LATIN CAPITAL LETTER GLOTTAL STOP	N/A		

Table 2. Glyphs required in different usage contexts

Consider the impact of these disunifications for font vendors or product vendors that include fonts with their products (e.g. operating systems, business-app suites). First, in the case of

qamats and *qamats qatan*, implementing support for TUS 4.1 is not a problem: the new character can be added to a font with no effect on existing documents. The revised font will be useful both for existing scenarios in which no distinction was made and also for new scenarios in which a two-way distinction is made.

In contrast, for the other two disunifications, there is no easy way for the change to be implemented.³ The glyphs for the existing characters cannot be changed in existing fonts without having potentially-damaging effects on existing documents. The new characters could be added to existing fonts, but because the glyphs for the existing characters cannot be changed, the result will be that both the existing and new characters have the same glyphs, which is not particularly useful.

Even in new fonts, which are not encumbered by legacy usage, there is no way to support both usage scenarios: in order to know what glyphs are needed for the existing characters, it must first be known whether the user does or doesn't make the two-way distinctions. The only real options are:

- create fonts that can only work for one usage scenario or the other; or
- create fonts that use the same default glyph for both existing and new characters with an alternate glyph for the existing character selectable by a font feature – but the two-way distinction will be available only in certain applications that support font-feature mechanisms.

For instance, after reviewing the disunification of *yerah ben yomo*, John Hudson (Tiro Typeworks) concluded that the best option for implementing the new character ATNAH HAFUKH was to use the same default glyph for both U+05AA YERAH BEN YOMO and U+05A2 ATNAH HAFUKH, and provide an alternate glyph for U+05AA for use when *galgal* is distinguished from *atnah hafukh*, selectable using an OpenType feature. John recently commented on this disunification on the Unicon list:⁴

“...the proposed disunification of *yerah ben yomo*... raises some problems at the display level, since in this case it is the existing character for which a glyph change would be required by users desiring to make the distinction visual... [This] is a problem we should have spotted when the new character was first proposed... But the fact that we failed to identify the problem early does not mean that the problem does not exist.

“My current inclination is to use the *etnah hafukh* glyph as default for both characters, and to handle the distinct form of *yerah ben yomo* as a glyph variant associated with a stylistic alternate feature. This is not ideal, since it requires a

³ The problem cannot be described as breaking *existing implementations*, since existing fonts can continue to be used in the same ways they were used before without any issues. Rather, the problem is that both of the post-disunification characters cannot be easily implemented, with potential for *new implementations*—revised or new fonts—to break existing documents.

⁴ Quoted from a message from John Hudson to the Unicon list, January 27, 2005, on the subject “QAMATS QATAN and HOLAM HASER FOR VAV”.

fairly sophisticated level of glyph substitution support from apps in order to handle what should be a fairly straight forward distinction between two characters.”

Some fonts are designed with specific uses in mind, and for such fonts the first option makes sense. This may be sufficient, for instance, for publishers of Hebrew religious texts who require a contrast between *galgal* and *atnah hafukh*. But this is the exceptional case: most users depend on fonts designed for general-purpose usage. Certainly for a platform vendor, such as Microsoft, fonts need to support as broad a range of uses as possible, and having to choose, for instance, between supporting phonetic transcription or the orthographies of living languages is a problem.

Avoiding the problems

It should be reasonably clear that the key factor that differentiates the *qamats* disunification from the other two is that it did not involve any change to the existing character, with only the new character requiring a different glyph. This was not the case with the other disunifications: the *yerah ben yomo* disunification involved a change in the representative glyph for U+05AA, and both resulted in a situation in which the existing character requires distinct glyphs depending on the usage.

In the case of *yerah ben yomo*, this could easily have been avoided by handling the disunification in a different way, as shown in Table 3:




Character	TUS 4.0	TUS 4.1
U+05AA HEBREW ACCENT YERAH BEN YOMO		
U+05xx HEBREW ACCENT GALGAL	N/A	

Table 3. Possible alternate disunification of *yerah ben yomo*

Reportedly, this alternative was considered by the proposers but abandoned since it would result in a less-than-ideal relationship between the name and glyph for U+05AA. End users are not the primary intended audience for character names, however, and less-than-ideal names can be mitigated by annotations or explanatory text in block descriptions. The cost of preserving the best possible name-glyph relationship has been the problems now faced by implementers, costs that will also be borne by end users.

It is too late to change the *yerah ben yomo* disunification, but the aforementioned problems associated with it can perhaps still be remedied by adding GALGAL as a second new character:





Character	TUS 4.0	TUS 5.0	Comment
U+05AA HEBREW ACCENT YERAH BEN YOMO			Used only in scenarios in which the two-way distinction is not made.
U+05A2 HEBREW ACCENT ATNAH HAFUKH	N/A		Used only in scenarios in which <i>atnah hafukh</i> is distinguished from <i>galgal</i> .
U+05xx HEBREW ACCENT GALGAL	N/A		

Table 4. Possible revised disunification of *yerah ben yomo*

This remedy to the current situation would have as a disadvantage that there would be two characters with the same glyph; in effect, one of the two characters would lose any useful purpose. That would simply have to be considered the price of having handled the initial disunification poorly. Arguably, this would be less problematic than the current situation since there are ways, at least, that the effective duplication can be dealt with in implementations, whereas there are no good ways for implementations to deal with the current situation.

The more important point, though, is that the need to create a situation in which one character becomes fully redundant could have been avoided in this case had there been a set of guiding principles for disunification in place beforehand.

The *glottal stop* disunification was a more difficult case. In terms of the glyphs needed for different usage contexts, it would have been adequate to make the existing character the capital in orthographic usage and add only one new character for the small glottal stop, but this was not a viable option because of problems related to changing the case property of the existing character. There was another alternative, though, which still remains as a possible remedy for the current problems: encode two new characters:

Character	TUS 4.0	TUS 5.0	Comment
U+0294 LATIN LETTER GLOTTAL STOP	?	?	Used only for phonetic transcription, or in orthographies without bi-cameral glottal stops.
U+0241 LATIN CAPITAL LETTER GLOTTAL STOP	N/A	?	Used only for orthographies that have bi-cameral glottal stops.
U+xxxx LATIN SMALL LETTER GLOTTAL STOP	N/A	?	

Table 5. Possible alternative / revised disunification of *glottal stop*

Again, there is a *visual* duplication of characters, though this duplication is only partial (unlike the situation that would hold for *yerah ben yomo* and *atnah hafukh*) since, in this case, the two characters would have distinct case properties. The visual duplication would be less than ideal, but it appears to be the only possible option that avoids the implementation problems described above.

In considering how these two disunifications could have been done without creating the implementation problems mentioned above, and how the disunifications can still be revised to remedy those problems, we find certain principles.

First, we do not want to disunify existing characters in a manner that entails changes to the glyphs of existing characters, since that is central to the problems that have been described.

More generally, we want to ensure that, as much as possible, viable uses of the existing character prior to disunification remain viable after the disunification.

In this regard, note that pre-disunification use of U+05AA specifically for *atnah hafukh*, with a private-use code point for the contrasting character *galgal*, would be viable, and this use of U+05AA would remain viable if a new character GALGAL were added. Thus, the possible disunification shown in Table 4 would have been workable. On the other hand, a pre-disunification use of U+0294 for an orthographic capital letter, with a private-use character for the lowercase counterpart, would not really have been viable because of the case property of U+0294; thus, it should not be essential to preserve that usage in a disunification of *glottal stop*.

Another principle we find is that, if it is not possible to add only one new character for the *distinct* letterform that is needed, then two new characters should be added, as that is the only way to create a distinction without creating implementation problems in relation to the existing character. This results in a visual or complete duplication of characters, and should be avoided if possible, but it must be recognized that there may be situations in which implementation problems cannot be avoided without such duplication. For instance, the existing character LATIN LETTER GLOTTAL STOP could not be used as the capital of a casing pair because its existing case property is *lowercase*, hence it was not possible to add only one new character for the x-height SMALL GLOTTAL STOP. The only way to create the distinction, then, without creating implementation problems in relation to the existing character is to add two new characters.

A further principle to note is that we must not give higher priority to a desire to have appropriate names for characters than we give to the impact on implementations. For instance, using the name YERAH BEN YOMO for *atnah hafukh* certainly would not be ideal, but that would be a much less serious concern than the problems now presented to implementers in how to support the existing and new characters.

Proposed principles on character disunification

In light of the preceding discussion, I propose that UTC adopt the following principles to be applied whenever a character disunification is being considered. These would be applied as general guidelines that could be overridden, but only after careful consideration.

When disunifying an existing character in the UCS, the following principles will be observed:

1. As much as possible, viable uses of existing characters prior to disunification will be preserved after disunification.
2. The normative properties of existing characters will not be changed.

3. The representative glyphs of existing characters will not be changed, and the range of glyphs expected for existing characters will not increase as a result of disunification.
4. If a character disunification cannot be achieved by adding one new character without requiring a change in normative properties of the existing character and without changing the representative glyph or range of expected glyphs for the existing character, then new characters will be added for each of the distinct, specific letterforms required; the existing character will not be intended for use in scenarios in which the distinct, specific letterforms are used. This may result in visually-duplicate characters, which in general should be avoided if possible, but may be necessary under the aforementioned conditions.
5. While it is desirable that a character name be fully appropriate to the given character and its representative glyph, concern over less-than-ideal names will not provide a sufficient basis for overriding principles 1 to 4, above.

Exceptions to these principles will be permitted only after careful consideration and on the basis of substantial rationale.

If these principles are accepted by UTC, I would further recommend that they be proposed for inclusion in WG2's *Principles and Procedures* document.

References

- Dotan, Aron, ed. 2001. *Biblia Hebraica Leningradensia*. Peabody, MA: Hendrickson Publishers.
- International Phonetic Association. 1999. *Handbook of the International Phonetic Association: A guide to the use of the International Phonetic Alphabet*. Cambridge: Cambridge University Press.
- Koyina, Laiza. 1983. *Dq weda goòle xè Teèt'o si. (The Blind Man and the Loon.)* Yellowknife, NWT, Canada: Northwest Territories Department of Education.
- Snaith, Norman Henry. 1982. *תורה נביאים וכתובים. (Hebrew Old Testament.)* London: British and Foreign Bible Society.



Unicode Standard Annex #31

Identifier and Pattern Syntax

Version	4.1.0
Authors	Mark Davis (mark.davis@us.ibm.com)
Date	2005-03-25
This Version	http://www.unicode.org/reports/tr31/tr31-5.html
Previous Version	http://www.unicode.org/reports/tr31/tr31-4.html
Latest Version	http://www.unicode.org/reports/tr31/
Revision	<u>5</u>

Summary

This document describes specifications for recommended defaults for the use of Unicode in the definitions of identifiers and in pattern-based syntax. It incorporates the Identifier section of Unicode 4.0 (somewhat reorganized) and a new section on the use of Unicode in patterns. As a part of the latter, it presents recommended new properties for addition to the Unicode Character Database. It also incorporates guidelines for use of normalization with identifiers (from UAX #15).

Section 2 supersedes Section 5.15 Identifiers from [[Unicode4.0](#)].

Section 5 supersedes Annex 7 in UAX #15: Normalization from [[Unicode4.0.1](#)].

Status

This document has been reviewed by Unicode members and other interested parties, and has been approved for publication by the Unicode Consortium. This is a stable document and may be used as reference material or cited as a normative reference by other specifications.

A Unicode Standard Annex (UAX) forms an integral part of the Unicode Standard, but is published 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 number of the Unicode Standard at the last point that the UAX document was updated.

Please submit corrigenda and other comments with the online reporting form [\[Feedback\]](#). Related information that is useful in understanding this document is found in [References](#). 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 [\[Versions\]](#).

Contents

- 1 [Introduction](#)
- 2 [Default Identifier Syntax](#)
- 3 [Alternative Identifier Syntax](#)
- 4 [Pattern Syntax](#)
- 5 [Normalization and Case](#)
- [Acknowledgements](#)
- [References](#)
- [Modifications](#)

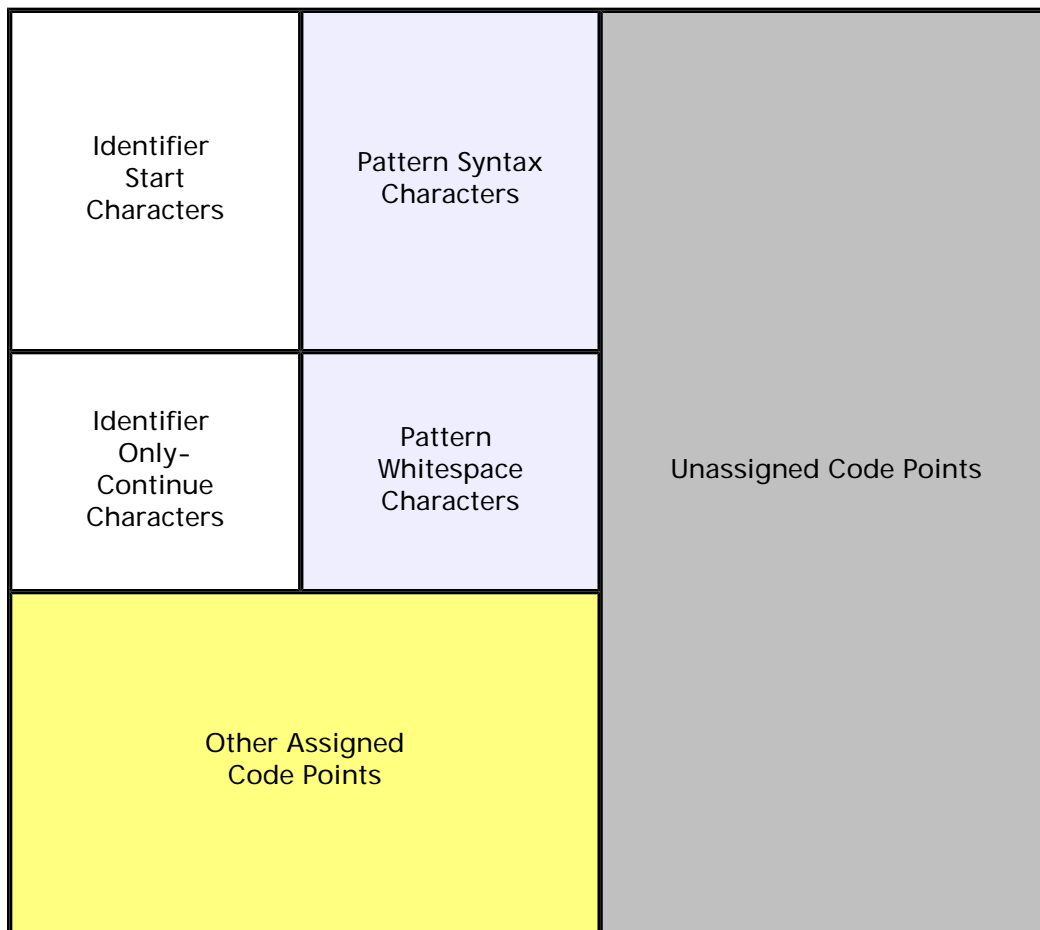
1 Introduction

A common task facing an implementer of the Unicode Standard is the provision of a parsing and/or lexing engine for identifiers. To assist in the standard treatment of identifiers in Unicode character-based parsers, a set of specifications is provided here as a recommended default for the definition of identifier syntax. These guidelines are no more complex than current rules in the common programming languages, except that they include more characters of different types. This document also provides guidelines for the user of normalization and case-insensitivity with identifiers, expanding on a section that was originally in UAX #15: Unicode Normalization Forms [\[UAX15\]](#).

These specifications provide a definition of identifiers that is guaranteed to be backward compatible with each successive release of Unicode, but also makes available any appropriate new Unicode characters. Unicode properties are also provided for stable pattern syntax: syntax that is stable over future versions of the Unicode Standard. These can either be used alone or with the identifier characters.

The following types of code points are defined (the sizes of the boxes are not to scale):

Character Classes for Programming



The set consisting of both Identifier Start and Only-Continue characters is known as Identifier Characters, also as Identifier Continue characters.

There are certain features that developers can depend on for stability:

Identifier characters, Pattern Syntax characters, and Pattern Whitespace are disjoint: they will never overlap.

The Identifier characters are always a superset of the Identifier Start characters

The Pattern Syntax characters and Pattern Whitespace characters are immutable, and will not change over successive versions of Unicode.

The Identifier characters and the Identifier Start characters may grow over time, either by the addition of new characters provided in a future version of Unicode, or (in rare cases) by the addition of characters that were in Other. However, neither will ever decrease.

In successive versions of Unicode, only the following changes are allowed, from one of the above classes to another:

Permitted Changes in Future Versions

	Identifier Start	Identifier Only Continue	Other Assigned
Unassigned	+	+	+
Other Assigned	+	+	
Identifier Only Continue	+		

The Unicode Consortium has formally adopted a stability policy on identifiers. For more information, see [\[Stability\]](#).

Each programming language standard has its own identifier syntax; different programming languages have different conventions for the use of certain characters such as \$, @, #, or _ in identifiers. To extend such a syntax to cover the full behavior of a Unicode implementation, implementers may combine those specific rules with the syntax and properties provided here.

That is, each programming language can define their identifier syntax as relative to the Unicode identifier syntax, such as saying that identifiers are defined by the Unicode properties, with the addition of "\$". By addition or subtraction of a small set of language specific characters, a programming language standard can easily track a growing repertoire of Unicode characters in a compatible way.

Similarly, each programming language can define white space characters or syntax characters relative to the Unicode pattern white space or syntax characters, with some specified set of additions or subtractions.

Systems that want to extend identifiers so as to encompass words used in natural languages may add characters identified in Section 4 Word Boundaries of [\[UAX29\]](#) with the property values Katakana, ALetter, and MidLetter, plus characters described in the notes at the end of that section.

Note that to preserve the disjoint nature of categories illustrated in the diagram "Character Classes for Programming", any character added to one of the categories must be subtracted from the others.

In some cases there are security implications that may require additional constraints on identifiers. For more information, see [\[UTR36\]](#).

1.1 Conformance

The following describes the possible ways that an implementation can claim conformance to this technical standard.

- C1. An implementation claiming conformance to this specification at any Level shall identify the version of this specification and the version of the Unicode Standard.
- C2. An implementation claiming conformance to Level 1 of this specification shall describe which of the following it observes:

[R1 Default Identifiers](#)

[R2 Alternative Identifiers](#)

[R3 Pattern Whitespace and Syntax Characters](#)

[R4 Normalized Identifiers](#)

[R5 Case-Insensitive Identifiers](#)

2 Default Identifier Syntax

The formal syntax provided here captures the general intent that an identifier consists of a string of characters beginning with a letter or an ideograph, and following with any number of letters, ideographs, digits, or underscores. It provides a definition of identifiers that is guaranteed to be backward compatible with each successive release of Unicode, but also adds any appropriate new Unicode characters.

D1. Default Identifier Syntax

`<identifier> := <ID_Start> <ID_Continue>*`

Identifiers are defined by the following sets of character categories from the Unicode Character Database.

Syntactic Classes for Identifiers

Properties	Alternates	General Description of Coverage
ID_Start	XID_Start	Uppercase letters, lowercase letters, titlecase letters, modifier letters, other letters, letter numbers, stability extensions
ID_Continue	XID_Continue	All of the above, plus nonspacing marks, spacing combining marks, decimal numbers, connector punctuations, stability extensions. These are also known simply as Identifier Characters, since they are a superset of the ID_Start . The set of ID_Start characters minus the ID_Continue characters are known as ID_Only_ Continue characters.

The innovations in the identifier syntax to cover the Unicode Standard include the following:

- Incorporation of proper handling of combining marks
- Allowance for layout and format control characters, which should be ignored when parsing identifiers
- The `XID_Start` and `XID_Continue` properties are alternates that incorporate the [NFKC Modifications](#).

2.1 Combining Marks

Combining marks are accounted for in identifier syntax: a composed character sequence consisting of a base character followed by any number of combining marks is valid in an identifier. Combining marks are required in the representation of many languages, and the conformance rules in Chapter 3, Conformance of [Unicode](#) require the interpretation of canonical-equivalent character sequences.

Enclosing combining marks (such as `U+20DD` and `U+20E0`) are excluded from the syntactic definition of `ID_Continue`, because the composite characters that result from their composition with letters are themselves not normally considered valid constituents of these identifiers.

2.2 Layout and Format Control Characters

Certain Unicode characters are used to control joining behavior, bidirectional ordering control, and alternative formats for display. These have the General Category value of Cf. Unlike space characters or other delimiters, they do not indicate word, line, or other unit boundaries.

While it is possible to ignore these characters in determining identifiers, the recommendation is to not ignore them, and not permit them in identifiers except in special cases. This is because of the possibility for confusion between two visually identical strings: see [UTR36](#). Some possible exceptions are the ZWJ and ZWNJ in certain contexts, such as between certain characters in Indic words.

2.3 Specific Character Adjustments

Specific identifier syntaxes can be treated as tailorings of the generic syntax based on character properties. For example, SQL identifiers allow an underscore as an identifier part, but not as an identifier start; C identifiers allow an underscore as either an identifier part or an identifier start. Specific languages may also want to exclude the characters that have a `decomposition_type` other than `canonical` or `none`, or to exclude some subset of those, such as those with a `decomposition_type` equal to `font`.

For programming language identifiers, normalization and case have a number of important implications. For a discussion of these issues, see [Normalization and Case](#).

2.4 Backward Compatibility

Unicode General Category values are kept as stable as possible, but they can change across versions of the Unicode Standard. The bulk of the characters having a given value are determined by other properties, and the coverage expands in the future according to the assignment of those properties. In addition, the `Other_ID_Start` property adds a small list of characters that qualified as `ID_Start` characters in some previous version of Unicode solely on the basis of their General Category properties, but that no longer qualify in the current version. In Unicode 4.1.0, this list consists of four characters:

U+2118	Script Capital P
U+212E	Estimated Symbol
U+309B	Katakana-Hiragana Voiced Sound Mark
U+309C	Katakana-Hiragana Semi-Voiced Sound Mark

Similarly, the `Other_ID_Continue` property adds a small list of characters that qualified as `ID_Continue` characters in some previous version of Unicode solely on the basis of their General Category properties, but that no longer qualify in the current version. In Unicode 4.1.0, this list consists of nine characters:

U+1369	ETHIOPIC DIGIT ONE
...	
U+1371	ETHIOPIC DIGIT NINE

The `Other_ID_Start` and `Other_ID_Continue` properties are thus designed to ensure that the Unicode identifier specification is backward compatible: Any sequence of characters that qualified as an identifier in some version of Unicode will continue to qualify as an identifier in future versions.

R1 Default Identifiers

To meet this requirement, an implementation shall use the `D1` and the properties `ID_Start` and `ID_Continue` (or `XID_Start` and `XID_Continue`) to determine whether a string is an identifier or not.

Or, it shall declare that it uses a modification, and provide a precise list of characters that are added to or removed from the above properties, and/or provide a list of additional constraints on identifiers.

3 Alternative Identifier Syntax

The disadvantage of working with the syntactic classes defined above is the storage space needed for the detailed definitions, plus the fact that with each new version of the Unicode Standard new characters are added, which an existing parser would not be able to recognize. In other words, the recommendations based on that table are not upwardly compatible.

This problem can be addressed by turning the question around. Instead of defining the set of code points that are allowed, define a small, fixed set of code points that are reserved for syntactic use and allow everything else (including unassigned code points) as part of an identifier. All parsers written to this specification would behave the same way for all versions of the Unicode Standard, because the classification of code points is fixed forever.

The drawback of this method is that it allows “nonsense” to be part of identifiers because the concerns of lexical classification and of human intelligibility are separated. Human intelligibility can, however, be addressed by other means, such as usage guidelines that encourage a restriction to meaningful terms for identifiers. For an example of such guidelines, see the XML 1.1 specification by the W3C [[XML1.1](#)].

By increasing the set of disallowed characters, a reasonably intuitive recommendation for identifiers can be achieved. This approach uses the full specification of identifier classes, as of a particular version of the Unicode Standard, and permanently disallows any characters not recommended in that version for inclusion in identifiers. All code points unassigned as of that version would be allowed in identifiers, so that any future additions to the standard would already be accounted for. This approach ensures both upwardly compatible identifier stability and a reasonable division of characters into those that do and do not make human sense as part of identifiers.

Some additional extensions to the list of disallowed code points can be made to further constrain “unnatural” identifiers. For example, one could include unassigned code points in blocks of characters set aside for future encoding as symbols, such as mathematical operators.

With or without such fine-tuning, such a compromise approach still incurs the expense of implementing large lists of code points. While they no longer change over time, it is a matter of choice whether the benefit of enforcing somewhat word-like identifiers justifies their cost.

Alternatively, one can use the properties described below, and allow all sequences of characters to be identifiers that are neither pattern syntax nor pattern whitespace. This has the advantage of simplicity and small tables, but allows many more “unnatural” identifiers.

[R2](#) Alternative Identifiers

To meet this requirement, an implementation shall define identifiers to be any string of characters that contains neither Pattern_White_Space nor Pattern_Syntax characters.

Or, it shall declare that it uses a modification, and provide a precise list of characters that are added to or removed from the sets of code points defined by these properties.

4 Pattern Syntax

There are many circumstances where software interprets patterns that are a mixture of literal characters, whitespace, and syntax characters. Examples include regular expressions, Java collation rules, Excel or ICU number formats, and many others. These patterns have been very limited in the past, and forced to use clumsy combinations of ASCII characters for their syntax. As Unicode becomes ubiquitous, some of these will start to use non-ASCII characters for their syntax: first as more readable optional alternatives, then eventually as the standard syntax.

For forward and backward compatibility, it is advantageous to have a fixed set of whitespace and syntax code points for use in patterns. This follows the recommendations that the Unicode Consortium made regarding completely stable identifiers, and the practice that is seen in XML 1.1 [\[XML1.1\]](#). (In particular, the consortium committed to not allocating characters suitable for identifiers in the range 2190..2BFF, which is being used by XML 1.1.)

With a fixed set of whitespace and syntax code points, a pattern language can then have a policy requiring all possible syntax characters (even ones currently unused) to be quoted if they are literals. By using this policy, it preserves the freedom to extend the syntax in the future by using those characters. Past patterns on future systems will always work; future patterns on past systems will signal an error instead of silently producing the wrong results.

Example:

In version 1.3 of program X, ' ' is a reserved syntax character, e.g. it does not perform an operation, but you have to quote it. In version 1.4, ' ' gets a real meaning, for example, "uppercase the subsequent characters". In program X, '\' quotes the next character; that is, causes it to be treated as a literal instead of a syntax character.

The pattern `abc...\ ...xyz` works on both version 1.3 and 1.4, and refers to the literal character since it is quoted in both cases.

The pattern `abc\... \...xyz` works on 1.1 and uppercases the following characters. On version 1.0, the engine (rightfully) has no idea what to do with `\`. Rather than silently fail (by ignoring or turning it into a literal), it has the opportunity signal an error.

As of Unicode 4.1.0, there are two Unicode character properties that can be used for stable syntax: `Pattern_White_Space` and `Pattern_Syntax`. Particular pattern languages may, of course, override these recommendations (for example, adding or removing other characters for compatibility in ASCII).

For stability, the property values are absolutely invariant; not changing with successive versions of Unicode. Of course, this does not limit the ability of the Unicode Standard to add more symbol or whitespace characters, but the syntax and whitespace characters recommended for use in patterns will not change.

When generating rules or patterns, all whitespace and syntax code points that are to be literals require quoting, using whatever quoting mechanism is available. For readability, it is recommended practice to quote or escape all literal whitespace and default ignorable code points as well.

Example: consider the following, where the items in angle brackets indicate literal characters.

`a<SPACE>b => x<ZERO WIDTH SPACE>y + z;`

Since `<SPACE>` is a `Pattern_White_Space` character, it requires quoting. Because `<ZERO WIDTH SPACE>` is a default ignorable character, it should also be quoted for readability. So if in this example `\uXXXX` is used for hex expression, but resolved before quoting, and single quotes are used for quoting, this might be expressed as:

`'a\u0020b' => 'x\u200By' + z;`

R3 Pattern Whitespace and Syntax Characters

To meet this requirement, an implementation shall use `Pattern_White_Space` characters as all and only those characters interpreted as whitespace in parsing, and shall use `Pattern_Syntax` characters as all and only those characters with syntactic use.

Or, it shall declare that it uses a modification, and provide a precise list of characters that are added to or removed from the sets of code points defined by these properties.

Note: all characters other than those defined by these properties would be available as identifiers or literals.

5 Normalization and Case

R4 Normalized Identifiers

To meet this requirement, an implementation shall specify the normalization form, and shall provide a precise list of any characters that are excluded from normalization, and if the normalization form is NFKC, shall apply the modifications in [NFKC Modifications](#) given by the properties `XID_Start` and `XID_Continue`. Except for identifiers containing excluded characters, any two identifiers that have the same normalization form shall be treated as equivalent by the implementation.

R5 Case-Insensitive Identifiers

To meet this requirement, an implementation shall specify either simple or full case folding, and adhere to the Unicode specification for that folding. Any two identifiers that have the same case-folded form shall be treated as equivalent by the implementation.

This section discusses issues that must be taken into account when considering normalization and case folding of identifiers in programming languages or scripting languages. Using normalization avoids many problems where apparently identical identifiers are not treated equivalently. Such problems can appear both during compilation and during linking, in particular across different programming languages. To avoid such problems, programming languages can normalize identifiers before storing or comparing them. Generally if the programming language has case-sensitive identifiers then Normalization Form C is appropriate, while if the programming language has case-insensitive identifiers then Normalization Form KC is more appropriate.

Note: In mathematically-oriented programming languages which make distinctive use of the Mathematical Alphanumeric Symbols such as [U+1D400 MATHEMATICAL BOLD CAPITAL A](#), an application of NFKC must filter characters to exclude characters with the property value `decomposition_type=font`. For related information, see [UTR #30: Character Foldings](#).

If programming languages are using NFKC to fold differences between characters, then they use the following modification of the identifier syntax from the Unicode Standard to deal with the idiosyncrasies of a small number of characters. These characters fall into three classes:

NFKC Modifications

1. Middle Dot. Because most Catalan legacy data will be encoded in Latin-1, `U+00B7 MIDDLE DOT` needs to be allowed in `ID_Continue`. (If the programming language is using a dot as an operator, then `U+2219 BULLET OPERATOR` or `U+22C5 DOT OPERATOR` should be used instead. However, care should be taken when dealing with `U+00B7 MIDDLE DOT`, as many processes will assume its use as punctuation, rather than as a letter extender.)
2. Combining-like characters. Certain characters are not formally combining characters, although they behave in most respects as if they were. Ideally, they should not be in `ID_Start`, but rather in `ID_Continue`, along with combining characters. In most cases, the

mismatch does not cause a problem, but when these characters have compatibility decompositions, they can cause identifiers not to be closed under Normalization Form KC. In particular, the following four characters are to be in ID_Continue and not ID_Start:

```
0E33 THAI CHARACTER SARA AM
0EB3 LAO VOWEL SIGN AM
FF9E HALFWIDTH KATAKANA VOICED SOUND MARK
FF9F HALFWIDTH KATAKANA SEMI-VOICED SOUND MARK
```

3. Irregularly decomposing characters. U+037A GREEK YPOGEGRAMMENI and certain Arabic presentation forms have irregular compatibility decompositions, and must be excluded from both ID_Start and ID_Continue. It is recommended that all Arabic presentation forms be excluded from identifiers in any event, although only a few of them must be excluded for normalization to guarantee identifier closure.

With these amendments to the identifier syntax, all identifiers are closed under all four Normalization forms. This means that for any string S,

```
isIdentifier(S) implies
    isIdentifier(toNFD(S))
    isIdentifier(toNFC(S))
    isIdentifier(toNFKD(S))
    isIdentifier(toNFKC(S))
```

Identifiers are also closed under case operations (with one exception), so that for any string S,

```
isIdentifier(S) implies
    isIdentifier(toLowercase(S))
    isIdentifier(toUppercase(S))
    isIdentifier(toFoldedcase(S))
```

The one exception is U+0345 COMBINING GREEK YPOGEGRAMMENI. In the very unusual case that U+0345 is at the start of S, U+0345 is not in ID_Start, but its uppercase and case-folded version are. In practice this is not a problem, because of the way normalization is used with identifiers.

Note: Those programming languages with case-insensitive identifiers should use the case foldings described in Section 3.13 Default Case Operations of [\[Unicode\]](#) to produce a case-insensitive normalized form.

When source text is parsed for identifiers, the folding of distinctions (using case mapping or NFKC) must be delayed until after parsing has located the identifiers. Thus such folding of distinctions should not be applied to string literals or to comments in program source text.

The UCD provides support for handling case folding with normalization: the property FC_NFKC_Closure can be used in case folding, so that a case folding of an NFKC string is itself normalized. These properties, and the files containing them, are described in the UCD documentation [[UCD](#)].

Acknowledgements

Thanks to Eric Muller, Asmus Freytag, and Martin Duerst for feedback on this document.

References

- | | |
|--------------|---|
| [Feedback] | Reporting Errors and Requesting Information Online
http://www.unicode.org/reporting.html |
| [Stability] | Unicode Consortium Stability Policies
http://www.unicode.org/standard/stability_policy.html |
| [Reports] | Unicode Technical Reports
http://www.unicode.org/reports/
For information on the status and development process for technical reports, and for a list of technical reports. |
| [UCD] | Unicode Character Database.
http://www.unicode.org/ucd/
For an overview of the Unicode Character Database and a list of its associated files |
| [Unicode] | The Unicode Standard
For the latest version see: http://www.unicode.org/versions/latest/ .
For the current version see: http://www.unicode.org/versions/Unicode4.1.0/ .
For the last major version see: The Unicode Consortium. The Unicode Standard, Version 4.0 . (Boston, MA, Addison-Wesley, 2003. 0-321-18578-1). |
| [Unicode4.0] | The Unicode Consortium. The Unicode Standard, Version 4.0 . Reading, MA, Addison-Wesley, 2003. 0-321-18578-1. |

[Unicode4.0.1]	The Unicode Consortium. The Unicode Standard, Version 4.0.1, defined by: The Unicode Standard, Version 4.0 (Boston, MA, Addison-Wesley, 2003. ISBN 0-321-18578-1), as amended by Unicode 4.0.1 (http://www.unicode.org/versions/Unicode4.0.1/).
[UAX15]	UAX #15: Unicode Normalization Forms http://www.unicode.org/reports/tr15/
[UAX29]	UAX #29: Text Boundaries http://www.unicode.org/reports/tr29/
[UAX36]	UTR #36: Security Considerations for the Implementation of Unicode and Related Technology http://unicode.org/reports/tr36/ in draft state, as of the publication of this document
[Versions]	Versions of the Unicode Standard http://www.unicode.org/versions/ For information on version numbering, and citing and referencing the Unicode Standard, the Unicode Character Database, and Unicode Technical Reports.
[XML1.1]	Extensible Markup Language (XML) 1.1 http://www.w3.org/TR/xml11/

Modifications

The following summarizes modifications from previous revisions of this document.

- 5 Removed section 4.1, since the two properties have been accepted for Unicode 4.1.
Expanded introduction
Adding information about stability, and tailoring for identifiers.
Added the list of characters in Other_ID_Continue .
Changed <identifier_continue> and <identifier_start> to just use the property names, to avoid confusion.
Included XID_Start and XID_Continue in R1 and elsewhere.
Added reference to UTR #36, and the phrase "or a list of additional constraints on identifiers" to R1.
Changed "Coverage" to "General Description of Coverage", since the UCD value are definitive.
Added clarifications in 2.4
Revamped 2.2 Layout and Format Control Characters
Minor editing
- 3 Made draft UAX
Incorporated Annex 7 from UAX #15
Added Other_ID_Continue for Unicode 4.1
Added conformance clauses
Changed <identifier_extend> to <identifier_continue> to better match the property name.
Some additional edits.
- 2 Modified Pattern White Space to remove compatibility characters
Added example explaining use of Pattern White Space
- 1 First version: incorporated section from Unicode 4.0 on Identifiers plus new section on patterns.

Copyright © 2000-2005 Unicode, Inc. All Rights Reserved. The Unicode Consortium makes no expressed or implied warranty of any kind, and assumes no liability for errors or omissions. No liability is assumed for incidental and consequential damages in connection with or arising out of the use of the information or programs contained or accompanying this technical report. The Unicode [Terms of Use](#) apply.

Unicode and the Unicode logo are trademarks of Unicode, Inc., and are registered in some jurisdictions.