

Proposed Draft Unicode® Technical Standard #58

UNICODE LINK DETECTION AND SERIALIZATION

Version	17.0 (draft 3)
Editors	Mark Davis
Date	2025-04-11
This Version	https://www.unicode.org/reports/tr58/tr58-1d3.html
Previous Version	https://www.unicode.org/reports/tr58/tr58-1d2.html
Latest Version	https://www.unicode.org/reports/tr58/
Latest Proposed Update	https://www.unicode.org/reports/tr58/proposed.html
Revision	1

Summary

There are flaws in certain ways that URLs are typically handled, flaws that substantially affect their usability for most people in the world — because most people's writing systems don't just consist of A-Z.

This document specifies two consistent, standardized mechanisms that address these problems, consisting of:

1. **link detection**: a mechanism for detecting URLs embedded in plain text that properly handles non-ASCII characters, and
2. **minimally escaping**: a mechanism for minimal escaping of non-ASCII code points in the Path, Query, and Fragment portions of a URL.

These two mechanisms are aligned, so that: a minimally escaped URL string between two spaces in flowing text is accurately detected, and a detected URL works when pasted into address bars of major browsers.

Status

This is a **draft** document which may be updated, replaced, or superseded by other documents at any time.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

A Unicode Technical Standard (UTS) is an independent specification. Conformance to the Unicode Standard does not imply conformance to any UTS.

Please submit corrigenda and other comments with the online reporting form [\[Feedback\]](#). Related information that is useful in understanding this document is found in the [References](#). For more information see [About Unicode Technical Reports](#) and the [Specifications FAQ](#). Unicode Technical Reports are governed by the Unicode [Terms of Use](#).

Contents

- 1 Introduction
- 2 Conformance
- 3 Link Detection Algorithm
- 4 Minimal Escaping
- 5 Email Addresses
- 6 Security Considerations
- 7 Property Data
- 8 Test Data
- 9 Migration
- Review Issues
- References
- Acknowledgments
- Modifications

Review Note: The Table of Contents will be fleshed out in a later draft.

1 Introduction

The standards for URLs and their implementations in browsers generally handle Unicode quite well, permitting people around the world to use their writing systems in those URLs. This is important: in writing their native languages, the majority of humanity uses characters that are not limited to A-Z, and they expect other characters to work equally well. But there are certain ways in which their characters fail to work seamlessly. For example, consider the common practice of providing user handles such as:

- x.com/rihanna
- bsky.app/profile/jaketapper.bsky.social
- www.instagram.com/vancityreynolds/
- www.youtube.com/@핑크퐁

[illegible]

and https://en.wikipedia.org/wiki/Antonín_Dvořák work fine as plain text; you can copy and paste them back into your address bar — they go to the right page and display properly in the address bar.

Notes

- Following *WHATWG URL: Goals*, this specification uses the term **URL** broadly, as including unescaped non-ASCII characters; that is, as utilizing the formal definition of IRIs. See also the W3C's *An Introduction to Multilingual Web Addresses*.
- In examples, links will be shown with a background color, to make the extent of the linkification clear.
- Serialization is the process of translating data into a format that can be stored or transmitted, and exactly reconstructed later. This document is concerned with serialization of a URL expressed in Unicode as people would see in an address bar into a readable textual form, *not* serialization into an internal format such as Punycode.

There is one other area that needs to be fixed in order to not treat non-English languages as second-class citizens. With most email programs, when someone pastes in the plain text:

- The page <https://ja.wikipedia.org/wiki/アルベルト・アインシュタイン> contains information about Albert Einstein.

and sends to someone else, they receive it as:

- The page <https://ja.wikipedia.org/wiki/アルベルト・アインシュタイン> contains information about Albert Einstein.

URLs are also “linkified” in many other applications, such when pasting into a word processor (triggered by typing a space afterwards, for example). However, many products (many text messaging apps, video messaging chats, etc.) completely fail to recognize any non-ASCII characters past the domain name. And even among those that do recognize such non-ASCII characters, there are gratuitous differences in where they *stop* linkifying.

Linkification is the process of adding links to URLs in plain text input, such as in emails, text messaging, or video meeting chats. The first step in this process is *link detection*, which is determining the boundaries of spans of text that contain URLs. That substring can then have a link applied to it in output text. The functions that perform these operations are called a *linkifier* and *link detector*, respectively.

The specifications for a URL don't specify how to handle link detection, since they are only concerned with the structure in isolation, not when it is embedded within flowing text. The lack of a clear specification for link detection also causes many implementations to overuse percent escaping for non-ASCII characters when converting URLs into plain text.

The linkification process for URLs is already fragmented — with different implementations producing very different results — but it is amplified with the addition of non-ASCII characters, which often have very different behavior. That is, developers' lack of familiarity with the behavior of non-ASCII characters has caused the different implementations of linkification to splinter. Yet non-ASCII characters are very important for readability. People do not want to see the above URL expressed in escaped ASCII:

- The page <https://ja.wikipedia.org/wiki/%E3%82%A2%E3%83%AB%E3%83%99%E3%83%AB%E3%83%88%29%E3%82%A2%E3%82%A4%E3%83%B3%E3%82%B7%E> contains information about Albert Einstein.

For example, take the lists of links on *List of articles every Wikipedia should have* in the available languages. When those are tested with major products, there are significant differences: any two implementations are likely to linkify those differently, such as terminating the linkification at different places, or not linkifying at all. That makes it very difficult to exchange URLs between products within plaintext, which is done surprisingly often — definitely causing problems for implementations that need predictable behavior.

This inconsistency causes problems for users and software companies. Having consistent rules for linkification also has additional benefits, leading to solutions for the following reported problems:

- If a system allows users to have their own user ids that end up in URLs, like <https://www.linkedin.com/in/my.user.name>, it can avoid user ids that have problematic linkification behavior, like trailing periods after path segments.
- Because linkification cannot be predicted for URLs with non-ASCII characters, common practice is to exchange them with escaped characters, which gives unreadable results such as the long line above.

If linkification behavior becomes more predictable across platforms and applications, applications will be able to do minimal escaping. For example, in the following only one character would need escaping, the %29 — representing an unmatched “)”:

- <https://ja.wikipedia.org/wiki/アルベルト%29アインシュタイン>

Providing a consistent, predictable solution that works well across the world's languages requires standardized algorithms to define the behavior, and the corresponding Unicode character properties covering all Unicode characters.

2 Conformance

UTS58-C1. For a given version of Unicode, a conformant implementation shall replicate the same link detection results as those produced by Section 3, *Link Detection Algorithm*.

UTS58-C2. For a given version of Unicode, a conformant implementation shall replicate the same minimal escaping results as those produced by Section 4, *Minimal Escaping*.

3 Link Detection Algorithm

The following table shows the relevant parts of a URL. For clarity, the separator characters are included in the examples. For more information see *WhatWG's URL: Example URL Components*.

Parts of a URL

Protocol	Host (incl. Domain)	Port	Path	Query	Fragment
https://	docs.foobar.com	:8000	/knowledge/area/	?name=article&topic=seo	#top

Note that the Protocol, Port, Path, Query, and Fragment are each optional.

Processes

There are two main processes involved in Unicode link detection.

- Initiation.** This requires determining the point within plaintext where the parsing of a URL starts. When the scheme is present for a URL (such as “http://”), determining the start of link detection is simple. However, the scheme for an URL is commonly omitted when URLs are represented in text. For example, the string “*adobe.com*” should be recognized as being an URL when it occurs in the body of an email message, even though it does not have a scheme.

2. **Termination.** This requires determining the point within plaintext where the parsing of a URL ends. A formal reading of the URL specs allows almost any character in certain fields, so it is insufficient for separating the end of the URL from the non-URL text after it.

Initiation

The start of a URL is easy to determine when it has a known protocol (eg, “https://”).

Implementations have also developed heuristics for determining the start of the URL when the protocol is elided, taking advantage of the fact that there are relatively few [top-level domains](#). And those techniques can be easily applied to internationalized domain names, which still have strong limitations on the valid characters. So the end of the domain name is also relatively easy to determine. For more information, see [UTS #46, Unicode IDNA Compatibility Processing](#)

The parsing up to the path, query, or fragment is as specified in *WHATWG URL*: [4.4. URL parsing](#).

For example, implementations must terminate link detection if a *forbidden host code point* is encountered, or if the host is a domain and a *forbidden domain code point* is encountered. Implementations must not linkify if a domain is not a *registrable domain*. The terms *forbidden host code point*, *forbidden domain code point*, and *registrable domain* are defined in *WHATWG URL*: [Host representation](#).

For example, an implementation would parse to the end of microsoft.com and google.de, foo.pð, or xn--j1ay.xn--p1ai.

Termination

Termination is much more challenging, because of the presence of characters from many different writing systems. While small, hard-coded sets of characters suffice for an ASCII implementation, there are over 150,000 Unicode characters, many with quite different behavior than ASCII. While in theory, almost any Unicode character can occur in certain fields in an URL, in practice many characters have very restricted usage in URLs.

Initiation stops at any Path, Query, or Fragment, so the termination process takes over with a “/”, “?”, or “#” character. Each Path, Query, or Fragment can contain most Unicode characters. The key is to be able to determine, given a Part (such as a Query), when a sequence of characters should cause termination of the link detection, even though that character would be valid in the URL specification.

It is impossible for a link detection algorithm to match user expectations in all circumstances, given the variation in usage of various characters both within and across languages. So the goal is to cover use cases as broadly as possible, recognizing that it will sometimes not match user expectations in certain cases. Exceptional cases (URLs that need to use characters that would terminate) can still be appropriately linkified if those few characters are represented with % escapes.

At a high level, this specification defines three features:

1. A method for identifying when to terminate link detection based on properties that define contexts for terminating the parsing of a URL.
 - This addresses the question, for example, when a trailing period should be counted as part of a link or not.
2. A method for identifying balanced quotes and brackets that enclose a URL
 - This addresses the distinction, for example, of enclosing the entire URL in parentheses, vs. URLs that contain a part that is enclosed in parens, etc.
3. An algorithm for doing the above, together with an enumerated property and a mapping.

One of the goals is also predictability; it should be relatively easy for users to understand the link detection behavior at a high level.

Properties

This specification defines two properties: [Link_Termination](#) (LTerm) and [Link_Paired_Opener](#) (LOpener).

Link_Termination Property

Link_Termination is an enumerated property of characters with five enumerated values: {**Include**, **Hard**, **Soft**, **Close**, **Open**}

Value	Description / Examples
Include	There is no stop before the character; it is included in the link.
	Example: <i>letters</i> <ul style="list-style-type: none">• https://ja.wikipedia.org/wiki/アルベルト・アインシュタイン
Hard	The URL terminates before this character.
	Example: <i>a space</i> <ul style="list-style-type: none">• Go to https://ja.wikipedia.org/wiki/アルベルト・アインシュタイン to find the material.
Soft	The URL terminates before this character, if it is followed by <code>/\p{Link_Termination=Soft}*(\p{Link_Termination=Hard} \$)/</code>
	Example: <i>a question mark</i> <ul style="list-style-type: none">• https://ja.wikipedia.org/wiki/アルベルト・アインシュタイン?abc• https://ja.wikipedia.org/wiki/アルベルト・アインシュタイン? abc• https://ja.wikipedia.org/wiki/アルベルト・アインシュタイン?
Close	If the character is paired with a previous character <i>in the same Part</i> (path, query, fragment) and in the same subpart (that is, not across interior "/" in a path, or across "&" or "=" in a query) , it is treated as Include . Otherwise it is treated as Hard .
	Example: <i>an end parenthesis</i> <ul style="list-style-type: none">• https://ja.wikipedia.org/wiki/(アルベルト)アインシュタインアインシュタイン• https://ja.wikipedia.org/wiki/アルベルト)アインシュタイン• https://ja.wikipedia.org/wiki/アルベルトアインシュタイン
Open	Used to match Close characters.
	Example: <i>same as under Close</i>

Link_Paired_Opener Property

Link_Paired_Opener is a string property of characters, which for each character in `\p{Link_Termination=Close}`, returns a character with `\p{Link_Termination=Open}`.

Review Note: Also see the [Review Issues](#).

Example

```
1. Link_Paired_Opener('}') == '{'
```

The specification of the characters with each of these property values is given in [Property Assignments](#).

Termination Algorithm

The termination algorithm assumes that a domain (or other host) has been successfully parsed to the start of a Path, Query, or Fragment, as per the algorithm in [WHATWG URL: 3. Hosts \(domains and IP addresses\)](#).

This algorithm then processes each final Part [path, query, fragment] of the URL in turn. It stops when it encounters a code point that meets one of the terminating conditions and reports the last location in the current Part that is still safely considered part of the link. The common terminating conditions are based on the Link_Termination and Link_Paired_Opener properties:

- A Link_Termination=Hard character, such as a space. Within a Path, “?” and “#” are handled as Hard. Within a Query, “#” is handled as Hard.
- A Link_Termination=Soft character, such as a ? that is followed by a sequence of zero or more Soft characters, then either a Hard character or the end of the text.
- A Link_Termination=Close character, such as a } that does **not** have a matching open character *in the same Part* of the URL. The matching process uses the Link_Paired_Opener property to determine the correct Open character, and matches against the top element of a stack of Open characters.

More formally:

The termination algorithm begins after the Host (and optionally Port) have been parsed, so there is potentially a Path, Query, or Fragment. In the algorithm below, each of those Parts has an initiator character, zero to two hard terminator characters, and zero to two clearStackOpen characters.

Part	initiator	terminators	clearStackOpen	Conditions
path	'/'	[?#]	[/]	
query	'?'	[#]	[=&]	
fragment (non-text)	'#'	[]	[]	
fragment (text)	'#'	[]	[-&.]	fragment starts with `#::~:~:text=`

Notes:

- `cp[i]` refers to the *i*th code point in the string being parsed, `cp[start]` is the first code point being considered, and *n* is the length of the string.
- For more information on text fragments, see [URL Fragment Text Directives](#).

Algorithm

1. Set lastSafe to 0 — *this marks the offset after the last code point that is included in the link detection (so far)*.
2. Set part to the Part whose initiator == cp[i]. If there is none, stop and return lastSafe.
3. Clear the openStack.
4. Loop from i = 0 to n - 1
 1. Set LT to Link_Termination(cp[i])
 2. **If part.clearStackOpen contains cp[i], clear the openStack.**
 3. If LT == Include
 1. If part.terminators contains cp[i]
 1. Set part to the Part whose initiator == cp[i]
 2. Clear the openStack.
 2. Set lastSafe to be i+1
 3. Continue loop
 4. If LT == Soft
 1. Continue loop
 5. If LT == Hard
 1. Stop and return lastSafe
 6. If LT == Open
 1. Push cp[i] onto openStack
 2. Set lastSafe to be i+1
 3. Continue loop.
 7. If LT == Open
 1. If openStack is empty
 1. Stop and return lastSafe
 2. Set lastOpen to the pop of openStack
 3. If Link_Paired_Opener(cp[i]) == lastOpen
 1. Set lastSafe to be i+1
 2. Continue loop.
 4. Else stop and return lastSafe.
5. After the loop terminates, return lastSafe.

For ease of understanding, this algorithm does not include all features of URL parsing.

The algorithm can be optimized in various ways, of course, as long as the results are the same.

Property Assignments

The draft property assignments are derived according to the following descriptions. **A full listing of the draft assignments supplied in Property Data.** Most characters that cause link termination would still be valid, but require % encoding.

Link_Termination=Hard

Whitespace, non-characters, **format, deprecated characters**, controls, private-use, surrogates, unassigned,...

- `[\p{whitespace} \p{NChar} [\p{C} - \p{Cf}] \p{deprecated}]`

Review Notes:

- The previous draft did not allow format characters, such as ZWJ, ZWNJ, TAGs, and so on.

Link_Termination=Soft

Termination characters and **ambiguous** quotation marks:

- `\p{Term}`
- `\p{lb=qu}`

Link_Termination=Open, Link_Termination=Close

Derived from `Link_Paired_Opener` property

Link_Termination=Include

All other code points

Link_Paired_Opener

if `BidiPairedBracketType(cp) == Close` then `Link_Paired_Opener(cp) = BidPairedBracket(cp)`

else if `cp == ">"` then `Link_Paired_Opener(cp) = "<"`

else `Link_Paired_Opener(cp) = \x{0}`

See [Bidi_Paired_Bracket](#).

4 Minimal Escaping

The goal is to be able to generate a serialized form of a URL that:

1. is correctly parsed by modern browsers and other devices
2. minimizes the use of percent-escapes
3. is completely link-detected when isolated.
 1. For example, "abc.com/path1./path2." would serialize as "abc.com/path./path2%2E" so that linkification will identify all of the serialized form within plaintext such as "See [abc.com/path./path2%2E](#) for more information".
 2. If not surrounded by Hard characters, the linkification may extend beyond the bounds of the serialized form. For example, "See [Xabc.com/path./path2%2EX](#) for more information".

Notes:

The minimal escaping algorithm is parallel to the linkification algorithm. Basically, when serializing a URL, a character in a Path, Query, or Fragment is only percent-escaped if it is: Hard, Close when unmatched, or Soft when it is the code point in the part.

In the following:

- `cp[i]` refers to the i^{th} code point in the `part` being serialized, `cp[0]` is the first code point in the `part`, and `n` is the number of code points.
- The algorithm assumes that the Path, Query, and Fragment have the normal interior escaping for syntactic characters such as the `part.terminators` and a `"/'` within `part` of a Path.
- A URL's internal model may contain bytes that arise from a page being in a legacy (non-UTF-8) character encoding. It is important, especially in the Query, to maintain those bytes even when they are invalid in UTF-8, such as `%FF` or `%C2%C2`. If the URL is known to originate in a page with a legacy character encoding (such as in an `href` value in that page), or is otherwise detected to have any invalid UTF-8 sequences, then an alternate serialization strategy should be used, such as percent-escaping each non-ASCII byte.

Minimal Escaping Algorithm

1. Set output to ""
2. Process each Part up to the Path, Query, and Fragment in the normal fashion, successively appending to output
3. For each `part` in any non-empty Path, Query, Fragment, successively:
 1. Append to output: `part.initiator`
 2. Set `copiedAlready = 0`
 3. Clear the `openStack`
 4. Loop from `i = 0` to `n - 1`
 1. If `part.terminators` contains `cp[i]`
 1. Set LT to Hard
 2. Else set LT to `Link_Termination(cp[i])`
 3. **If `part.clearStackOpen` contains `cp[i]`, clear the `openStack`.**
 4. If LT == Include
 1. Append to output: any code points between `copiedAlready` (inclusive) and `i` (exclusive)
 2. Append to output: `cp[i]`
 3. Set `copiedAlready` to `i+1`
 4. Continue loop
 5. If LT == Hard
 1. Append to output: any code points between `copiedAlready` (inclusive) and `i` (exclusive)
 2. Append to output: `percentEscape(cp[i])`
 3. Set `copiedAlready` to `i+1`
 4. Continue loop
 6. If LT == Soft

1. Continue loop
7. If `LT == open`
 1. Push `cp[i]` onto `openStack`
 2. Do the same as `LT == Include`
8. If `LT == Close`
 1. Set `lastOpen` to the pop of `openStack`, or 0 if the `openStack` is empty
 2. If `Link_Paired_Opener(cp[i]) == lastOpen`
 1. Do the same as `LT == Include`
 3. Else do the same as `LT == Hard`
5. If part is not last
 1. Append to output: all code points between `copiedAlready` (inclusive) and `n` (exclusive)
6. Else if `copiedAlready < n`
 1. Append to output: all code points between `copiedAlready` (inclusive) and `n - 1` (exclusive)
 2. Append to output: `percentEscape(cp[i])`
4. Return output.

The algorithm can be optimized in various ways, of course, as long as the results are the same. For example, the interior escaping for syntactic characters can be combined into a single pass.

Additional characters can be escaped to reduce confusability, especially when they are confusable with URL syntax characters, such as a `?` character in a path. See [Security Considerations](#) below.

5 Email Addresses

Email address link detection applies similar principles. An email address is of the form `local-part '@' domain-name`. The algorithm is invoked whenever an `@` character is encountered at index `n`.

1. Let `disallowedAscii` be the set consisting of ASCII control characters, space, and any of: `.,:;()[]{}@ \ < >`.
2. Scan forward from `n+1` to determine if the `@` sign is followed by a valid domain name (terminating at index end).
3. If there is no such valid domain name, then return a failure code indicating that there was no email address containing that `@`.
4. If there is such a valid domain name, scan backward through the text from `i = n - 1` down to `-1`
 1. If `i < 0`, set `start = 0` and terminate scanning.
 2. Else if `cp[i]` is in `disallowedAscii`, set `start = 0` and terminate scanning.
 3. Else if `Link_Termination(cp[i]) != Include`, set `start = i + 1` and terminate scanning.
 4. Else continue scanning backwards.
5. If `start == n`, then return a failure code indicating that there was no email address containing that `@`.
6. Else return the pair `start, end`.

This can be adapted to produce an algorithm that is single-pass, as long as it produces the same results.

Review Note: Flesh out the above more precisely, and describe how to handle quoted email addresses, and the special cases of `.` at the start or end of the `local-part`, or as `".."`

6 Security Considerations

The security considerations for Path, Query, and Fragment are far less important than for Domain names. See [UTS #39: Unicode Security](#) for more information about domain names.

There are documented cases of how Format characters can be used to sneak malicious instructions into LLMs; see [Invisible text that AI chatbots understand and humans can't?](#) URLs are just a small part of the larger problem of feeding *clean text* to LLMs, both in building them and in querying them: making sure the text does not have malformed encodings, is in a consistent Unicode Normalization Form (NFC), and so on.

For security implications of URLs in general, see [UTS #39: Unicode Security Mechanisms](#). For related issues, see [UTS #55 Unicode Source Code Handling](#). For display of BIDI URLs, see also [HL4 in UAX #9, Unicode Bidirectional Algorithm](#).

7 Property Data

The draft assignments of `Link_Termination` and `Link_Paired_Opener` property values are in the following locations **while this document is in draft form. The final locations would be in <https://unicode.org/Public/links/>.**

- `LinkTermination.txt`
- `LinkPairedOpener.txt`

Review Note: An `@missing` will be added to `LinkTermination.txt`, and `LinkPairedOpener.txt` will have a pointer to the definition of `BidiPairedBracket`.

Review Note: For comparison to the related `General_Category` values, see the characters in:

- `(Close_Punctuation + Final_Punctuation - BidiPairedBracketType=Close)`
- `(Initial_Punctuation + Open_Punctuation - BidiPairedBracketType=Open)`

8 Test Data

The format for test files is not yet settled, but the files might look something like the following. **The final locations would be in <https://unicode.org/Public/links/>.**

- `LinkificationTest.txt`
- `SerializationTest.txt`

Review Note: The plan is to add a file of Wikipedia URLs for testing.

Migration

Migration: Link Detection

1. Call the existing code library.
2. Let S be the start of the link in plain text as detected by the existing code library, and E be the offset at the end of that link.
3. If E is at the end of the string, or if the code point following E has the value Link_Termination=Hard, then return S and E.
4. Scan backwards to find the last initiator ([/?#]).
5. Follow the [Termination Algorithm](#) from that point on.

Review Issues

There is a further complication, that some quotation marks appear in non-paired usage, such as RIGHT SINGLE QUOTATION MARK or APOSTROPHE, but also QUOTATION MARK as an alternative to HEBREW PUNCTUATION GERSHAYIM. The simplest and most predictable solution is to have them be Soft.

The < and > characters are added to Link_Paired_Opener to set off URLs, such as <<https://eel.is/c++draft/vector.bool.psp#lib:vector<bool>>> and <<https://wg21.link/p2348>>. While many sources that formerly recommended that practice no longer do (such as the Chicago Manual of Style), others have continued the practice, such as in C++ sg16.

TBD

Thanks to the following people for their contributions and/or feedback on this document: Dennis Tan, Erika Etemad, Hayato Ito, Markus Scherer, Mathias Bynens, Robin Leroy, **TBD flesh out further**

The following summarizes modifications from the previous revision of this document.

- Modified the [Termination Algorithm](#) to handle [scroll-to-text fragments](#).
- Added a new section 5 [Email Addresses](#).
- Made various small modifications based on UTC comments.
- Fixed 4.7 to be `If LT == Close`
- Added review notes for intended future changes: the location of data files (properties and test data), etc.

- Changed the title, summary, and introduction to reflect the fact that this is concerned with more than linkification, and is really centered around usability for languages that use characters beyond A-Z.
- Link_Termination=Close characters: modified the behavior to require matching to not be across interior '/' in a path, or across '&'; or '=' in a query. As a part of this, added clearStackOpen to the Part information.
- Link_Termination=Hard characters: removed Cf (Format) characters and added deprecated characters.
- Link_Termination=Soft characters: replaced the hard-coded list ['<' '>' '""' '-' '«' '»'] by the property-based {p{lb=qu}. This broadens the set slightly (with relatively rare characters), but also makes it more robust.
- Added migration section, to discuss how the changes could be deployed on top of existing library calls.
- Added draft acknowledgments.
- Various copy-edits

- Problematic links were unlinked (they still have a highlight, but aren't active)
- Added the 2nd conformance clause in [Conformance](#)
- Fleshed out [Minimal Escaping](#)
- Made a substantive fix to the [Termination Algorithm](#) (to "If LT == Open").
- Fleshed out the review note in [Security](#) to be more specific about the contexts for the two examples mentioned (ZWJ/ZWNJ, and TAG characters), and add a note about matching brackets across syntax characters.
- Added draft samples in [Test Data](#)
- Various copy-edits

Modifications for previous versions are listed in those respective versions.

© 2024–2024 Unicode, Inc. This publication is protected by copyright, and permission must be obtained from Unicode, Inc. prior to any reproduction, modification, or other use not permitted by the [Terms of Use](#). Specifically, you may make copies of this publication and may annotate and translate it solely for personal or internal business purposes and not for public distribution, provided that any such permitted copies and modifications fully reproduce all copyright and other legal notices contained in the original. You may not make copies of or modifications to this publication for public distribution, or incorporate it in whole or in part into any product or publication without the express written permission of Unicode.

Use of all Unicode Products, including this publication, is governed by the Unicode [Terms of Use](#). The authors, contributors, and publishers have taken care in the preparation of this publication, but make no express or implied representation or warranty of any kind and assume no responsibility or liability for errors or omissions or for consequential or incidental damages that may arise therefrom. This publication is provided "AS-IS" without charge as a convenience to users.

Unicode and the Unicode Logo are registered trademarks of Unicode, Inc. in the United States and other countries.