Proposal for Prefix ZW Operators

nick black, consulting scientist
nickblack@linux.com

July 11, 2021

Abstract

When processing a stream of Unicode, there are several cases where it is useful or even necessary to know that a codepoint is only part of a ZWJ sequence. This is not possible with existing ZWJ semantics, since the ZWJ arrives only after each non-terminating combined character. I propose stacking, prefixing ZWJ and ZWNJ, indicating that the subsequent two encoded codepoints and/or ZWJ sequences are to be combined into a single ZWJ sequence. Backwards compatibility is maintained.

1 Introduction

As the author of a modern library for TUIs and character graphics[1], I've spent significant time over the past two years working with complex Unicode in interactive environments. Interactivity, and more fundamentally streaming, adds complexity to handling Unicode, especially with regard to combining characters, ZWJ sequences[2], and segmentation[3]. The entirety of an EGC or ZWJ sequence might not be available when the non-terminating character(s) are read; in the interactive or simply highly latent case, subsequent elements of the sequence might not be available even after user-perceivable delay. It is even possible that subsequent elements are never received. What is a program to do with the base characters in the interim?

The widespread UTF-8 encoding[5] allows an encoded character's length to be determined from its first byte. Assuming a byte-oriented transport, there is thus never any question as to whether the entire encoded character has been received. The (non-trivial) prefix of a given encoded character cannot be the prefix of any other result of valid UTF-8 encoding. EGCs and ZWJ sequences do not enjoy this happy property; many (but not all) prefixes of these text elements are themselves valid standalone EGCs.

2 The Valid Prefix Problems

The most fundamental problem due to valid prefixes is that **something sensible can be done with the prefix in isolation**. This is not possible when reading UTF-8; if we receive an incomplete encoded character, this fact is obvious. If we receive a message ending in such an incomplete character, we know that it was truncated; the user can be informed of the truncation, and certainly no incorrect glyph will be substituted. There need be no feedback to a typing user, as nothing they might type would generate such an incomplete output.

This is not true for EGCs or ZWJ sequences. Most catastrophically, a truncation can change the entire meaning of a sequence, replacing it with a different (but semantically viable) concept. For instance, imagine a passenger liner has received the message "Heads up! Your path will pass by 🏴". It is not unreasonable

to assume that the passengers will be instructed to bring out their cameras, in the hope of seeing the New Zealand All-Blacks and one of their mighty *haka* dances. In truth, of course, they ought have drawn their ⚔s; the missing U+200D and U+2620 would have indicated the presence of pirates 🏴‍☠️. Were I to describe the action of *Indiana Jones and the Temple of Doom* as "In the hand of Mola Ram is held the victim's ❤️‍🔥", that's a completely different subtext than "🔥". A manager delegating a task to her Sudanese mermaid subordinate might address Aarifa as 🧜🏾 ("mermaid: medium-dark skin tone"); should that be truncated to 🧜 ("merperson"), she's not only created a non-inclusive environment for BIPOC merpeople, but inadvertently misgendered Aarifa, now a crime in several jurisdictions (not to mention possibly confusing any non-binary merployees). This is no way to solve tech's diversity problem.

There are further issues. In an interactive environment, typing a ZWJ sequence usually involves entering a series of characters. What is to be displayed following the first valid prefix? If the valid prefix happened to be wider than the resulting sequence[1], it could lead to a false line wrap. A glyph similar to (but distinct from) the sequence glyph might be confusing, and catalyze errors. Alternatively, an editor might wish to indicate that further input is desired; this is not possible with infix ZWJ. Several threads might pull encoded characters from a common buffer; while a thread can be certain it is atomically pulling a complete UTF-8 character, it cannot be certain that it has pulled the entirety of an EGC or ZWJ sequence. The choice is then either to wait an arbitrary amount of time for further possible characters whenever a valid prefix is received (not allowing earlier characters to be processed), or for some other thread to retrieve the suffix, and likely throw an error.

3 Resolution via Prefix ZWJ

For EGCs made up of combining characters, there is no general way to solve this problem, but for ZWJs with their explicit binding control, a solution is at hand: introduce the bound sequence with a *prefix* ZWJ, henceforth known as PZWJ. Like the infix ZWJ, PZWJ always works on two text elements. Like the infix ZWJ, PZWJ can be chained to use several component characters. Unlike the infix ZWJ, PZWJ completely and unambiguously describes the resulting structure *from the beginning of the structure*. Concatenation is associative[4], and thus all possible distributions of $N-1$ PZWJs across N characters yield the same result.

4 Details of PZWJ

PZWJ and PZWNJ will live in the Supplemental Punctuation block, occupying the unused codepoints `U+2E5C` and `U+2E5D`, with the names `Prefix Zero Width Joiner` and `Prefix Zero Width Non-Joiner`. The least significant hex digits have been chosen to match ZWJ/ZWNJ.

PZWJ binds more tightly than infix ZWJ. If PZWJ binds less tightly than ZWJ, it defeats the entire point. Right-associativity, as much as it applies at all, follows directly from prefix notation. It is expected that text making use of PZWJ will forego ZWJ entirely, but a PZWJ sequence may form the right-hand side of a ZWJ sequence. It **must not** form the left-hand side of a ZWJ sequence (this would defeat the point), though it can form the right-hand side. A PZWJ sequence can form one or both sides of another PZWJ sequence. Canonicalization will continue to admit ZWJ sequences, but any PZWJ sequence **must** canonicalize as PZWJs only, built up entirely using the form

```
PZWJ {base character} {PZWJ sequence or base character}
```

This allows canonicalizing mixed ZWJ and PZWJ intro pure PZWJ in O(1) space, in a single pass, from either left-to-right or right-to-left. All Unicode ZWJ sequences can thus be unambiguously and mechanically rewritten as canonical PZWJ sequences.

PZWJ and PZWNJ have equivalent precedence.

If a stream terminates without providing all necessary elements of a PZWJ sequence, this can be detected. The mutilated PZWJ sequence can be passed through, since it cannot be interpreted as a valid form. The

[1] I do not believe this to be possible through Unicode 13.1.

presence of such a malformed trailer **should** be indicated to the user via some means.

5 Contact Info

nick black
Dirty South Supercomputers and Waffles, LLC
855 Peachtree St. NE
Suite 3204
Atlanta, GA 30308

References

[1] nick black. *Hacking the Planet (with Notcurses): A Guide to Character Graphics and TUIs*. 1st ed. Kindle Direct Publishing, 2020. ISBN: 9798620069491.

[2] The Unicode Consortium. *Unicode® Data Files: Emoji ZWJ Sequences*. Tech. rep. Mountain View, CA, 2020.

[3] The Unicode Consortium. *Unicode® Standard Annex #29: Unicode Text Segmentation*. Tech. rep. Mountain View, CA, 2020.

[4] John Hopcroft and Jeffrey Ullman. *Introduction to Automata Theory, Languages, and Computation*. 1st ed. Addison-Wesley, 1979. ISBN: 9780321455369.

[5] F. Yergeau. *UTF-8, a transformation format of ISO 10646*. STD 63. http://www.rfc-editor.org/rfc/rfc3629.txt. RFC Editor, 2003-11. URL: http://www.rfc-editor.org/rfc/rfc3629.txt.