L2/17-197

Request to Retract Consensus 151-C19 and Action Item 151-A134

Henri Sivonen, Mozilla (hsivonen@mozilla.com)
2017-06-12

Please retract consensus 151-C19 and action item 151-A134 for reasons given in the write-up copied below—primarily because the guidance in section "Best Practices for Using U+FFFD" of the Unicode Standard in its form prior to 151-C19 has been adopted in multiple prominent implementations and the prominent non-ICU implementations tested that do not implement the pre-151-C19 guidance exactly are still closer to the pre-151-C19 guidance than the guidance proposed in L2/17-168.

The write-up copied below, https://hsivonen.fi/broken-utf-8/, was shared in its original form on the Unicode Mailing List on 2017-05-31. Deletions since then have been marked with strike-through and additions with underline. The test input file is available at https://hsivonen.fi/broken-utf-8/test.html

How Many REPLACEMENT CHARACTERs?

The Unicode Technical Committee recently decided to change their long-standing guidance for the preferred number of REPLACEMENT CHARACTERs generated for bogus byte sequences when decoding UTF-8. I think this change is inappropriate because it was based on mere aesthetic considerations and ICU's behavior and goes against the behavior of multiple prominent existing implementations that implemented the long-standing previous guidance.

Background

Not all byte sequences are valid UTF-8. When decoding potentially invalid UTF-8 input into a valid Unicode representation, something has to be done about invalid input. One approach is to stop altogether and to signal an error upon finding invalid input. While this is a valid response for some applications, it is not our topic today. The topic at hand is what to do in the non-Draconian case where the decoder continues even after discovering invalid input.

The naïve answer is to ignore invalid input until finding valid input again (i.e. finding the next byte that has a lead-byte value), but this is dangerous and should never be done. The danger is that silently dropping bogus bytes might make a string that didn't look dangerous with the bogus bytes present become valid active content. Most simply, `<scr�ipt>` (� standing in for a bogus byte) could become `<script>` if the error is ignored. So it's non-controversial that every sequence of bogus bytes should result in *at least one* REPLACEMENT CHARACTER and that the next lead-valued byte is the first byte that's no longer part of the invalid sequence.

But how many REPLACEMENT CHARACTERs should be generated for a sequence of multiple bogus bytes?

Unicode 9.0.0 (page 127) says: "An ill-formed subsequence consisting of more than one code unit could be treated as a single error or as multiple errors. For example, in processing the UTF-8 code unit sequence <F0 80 80 41>, the only formal requirement mandated by Unicode conformance for a converter is that the <41> be processed and correctly interpreted as <U+0041>. The converter could return <U+FFFD, U+0041>, handling <F0 80 80> as a single error, or <U+FFFD, U+FFFD, U+FFFD, U+0041>, handling each byte of <F0 80 80> as a separate error, or could take other approaches to signalling <F0 80 80> as an ill-formed code unit subsequence." So *as far as Unicode is concerned*, any number from one to the number of bytes in the number of bogus bytes (inclusive) is OK. In other words, the precise number is implementation-defined *as far as Unicode is concerned.*

Yet, immediately after saying that there isn't conformance requirement for the precise number, the Unicode Standard proceeds to express a *preference* motivating it by saying: "To promote interoperability in the implementation of conversion processes, the Unicode Standard recommends a particular best practice." The "best practice" (until the recent change) was that a maximal invalid sequence of bytes that forms a prefix of a valid sequence is collapsed into one REPLACEMENT CHARACTER and otherwise there is one REPLACEMENT CHARACTER per each bogus byte.

Explaining the Old Preference in the Terms of Implementation

The old preference makes sense when the UTF-8 to decoder is viewed as a state machine that recognizes UTF-8 as a regular grammar based on the information presented in table 3-7 "Well-Formed UTF-8 Byte Sequences" in the Unicode Standard (page 125 in version 9.0.0; quoted below) and exhibits the following behavior when encountering a byte that doesn't fit the grammar at the current state:

- If the state machine is in the start state, consume the bogus byte (which is never a lead byte since lead bytes are allowed in the start state) and emit a REPLACEMENT CHARACTER.
- If the state machine is *not* in the start state, unconsume the bogus byte (which may be a lead byte), change state to the start state (i.e. the byte will be reprocessed in the start state shortly) and emit a REPLACEMENT CHARACTER.

The conclusion here is that when viewing the UTF-8 decoder as a state machine that encodes knowledge of what byte sequences are valid, the old preference makes perfect sense. In particular, the rule to collapse prefixes of valid sequences is not added complexity but the simple thing arising from not requiring the state machine to unconsume more than the one byte under examination.

Table 3-7. Well-Formed UTF-8 Byte Sequences

Code Points	First Byte	Second Byte	Third Byte	Fourth Byte
U+0000..U+007F	00..7F			

Code Points	First Byte	Second Byte	Third Byte	Fourth Byte
U+0080..U+07FF	C2..DF	80..BF		
U+0800..U+0FFF	E0	*A0*..BF	80..BF	
U+1000..U+CFFF	E1..EC	80..BF	80..BF	
U+D000..U+D7FF	ED	80..*9F*	80..BF	
U+E000..U+FFFF	EE..EF	80..BF	80..BF	
U+10000..U+3FFFF	F0	*90*..BF	80..BF	80..BF
U+40000..U+FFFFF	F1..F3	80..BF	80..BF	80..BF
U+100000..U+10FFFF	F4	80..*8F*	80..BF	80..BF

The New Preference

On May 12 2017, the Unicode Technical Committee accepted a proposal (for Unicode 11) to collapse sequences of bogus bytes to a single REPLACEMENT CHARACTER not only when they form a prefix of a valid sequence but also when the bogus bytes fit as a prefix of the general UTF-8 bit pattern. The bit pattern for one, two, three and four-byte sequences is given in table 3-6 “UTF-8 Bit Distribution” in the Unicode Standard (page 125 in version 9.0.0; quoted below). The proposal is ambiguous about whether to do the same thing for five and six-byte sequences whose bit pattern is not defined as existing in Unicode but was defined in now-obsolete RFCs for UTF-8, the last RFC defining them being RFC 2279.

If five and six-byte sequences are treated according to the logic of the newly-accepted proposal, the newly-accepted proposal matches the behavior of ICU. If the decoder is supposed to be unaware of five and six-byte patterns, which are non-existent as far as Unicode is concerned, I am not aware of any implementation matching the new guidance.

The rationale against the old guidance was “I believe the best practices are wrong” and the rationale in favor of the new guidance was “feels right”. (Really.)

Table 3-6. UTF-8 Bit Distribution

Scalar Value	First Byte	Second Byte	Third Byte	Fourth Byte
00000000 0xxxxxxx	0xxxxxxx			
00000yyy yyxxxxxx	110yyyyy	10xxxxxx		
zzzzyyyy yyxxxxxx	1110zzzz	10yyyyyy	10xxxxxx	
000uuuuu zzzzyyyy yyxxxxxx	11110uuu	10uuzzzz	10yyyyyy	10xxxxxx

Explaining the New Preference in Terms of Implementation

The new preference makes sense if the UTF-8 decoder is viewed as a bit accumulator that first consumes bytes according to the UTF-8 bit distribution pattern and masks and shifts the variable bits into an accumulator where they form a scalar value and then upon completing a sequence according to the bit distribution pattern checks if the scalar value is valid given the length of sequence consumed. Scalar value for the surrogate range or above the Unicode range is always invalid and otherwise scalar values are invalid if the scalar value could be represented as a shorter sequence of bytes than a sequence that was actually consumed.

It is worth noting that the concept of accumulating a scalar value during UTF-8 decoding is biased towards using UTF-16 or UTF 32 as the in-memory Unicode representation, since decoding to those forms necessarily involves the use of such an accumulator. When decoding UTF-8 to UTF-8 as the in-memory Unicode representation, while it is possible to first accumulate the scalar value and then re-encode it as UTF-8, it is unnecessary and inefficient and the sort of validation state machine described above makes more sense. Sure, such a state machine could be extended to exhibit the outward behavior of the formulation that involves a scalar accumulator, but it would be extra complexity in service of replicating the behaviors arising from a different model.

What's Wrong with Changing a Mere Preference?

So who cares? It's just a non-normative expression of preference.

There are multiple reasons to care.

- There are multiple prominent implementations that follow the old guidance and it's wrong to make them explain themselves (why they do not follow the new preference) or, worse, change behavior risking the introduction of bugs.
- It sets a bad precedent to change the Unicode Standard instead of changing the Unicode Consortium-hosted implementation (ICU) when the two disagree but multiple prominent implementations follow the Standard and ICU is the outlier.
- The old guidance isn't a mere preference but a conformance requirement in the Web context.
- The change was dodgy in terms of committee process, which sets a bad precedent.
- Chrome has had a bug arising from different error handling behavior between two UTF-8 decoder implementations in the codebase.

If anything "It's not a *requirement*." should be taken as an argument why the spec doesn't need changing and not as an argument why changes on flimsy grounds are OK. The realization that the Unicode Consortium seems to lack a strong objective reasoning to prefer a particular number of REPLACEMENT CHARACTERs could be taken to support a conclusion that maybe it would be the best for the Unicode Standard not to express a preference (called "best practice" implying the preference is in some sense the "best" option) on this topic, but it does not support the conclusion that changing the expressed behavior whichever way is OK.

But most importantly, even if the issue of the exact number of REPLACEMENT CHARACTERs in itself was not really that important to care about and it might seem silly to write this much about it, I see changing a widely-implemented spec on flimsy grounds as poor standard stewardship and I wish the Unicode Consortium did better than that so that this kind of failure to investigate what multiple implementations do does not repeat with something more important.

What Do Implementations Do, Then?

I tested the following implementations (Web browsers by visual inspection and others by performing the conversion and using `diff` on the output):

- Firefox 53.0.3 (uconv conversion library)
- Chrome 58.0.3029.110 (uses Web Template Framework forked from WebKit but since then modified for UTF-8 despite using ICU for many legacy encodings)
- Safari 10.1.1 on macOS 10.11.6 and the lastest release version of Safari on macOS 10.12.5 (uses Web Template Framework for UTF-8 despite using ICU for many legacy encodings)
- Edge (EdgeHTML 14.14393 on Windows 10 1607)
- IE11 (on Windows 10 1607)
- ICU (55.1 as distributed on Ubuntu 16.04)
- Win32 (`MultiByteToWideChar` on Windows 10 1607)
- The Java standard library as implemented in OpenJDK 8 (1.8.0_131 as distributed on Ubuntu 16.04)
- The Ruby standard library (2.3.1p112 as distributed on Ubuntu 16.04)
- The Python 3 standard library (3.5.2 as distributed on Ubuntu 16.04)
- The Python 2 standard library (2.7.12 wide build as distributed on Ubuntu 16.04)
- The Python 2 standard library (2.7.10 narrow build as distributed on macOS 10.11.6)
- The Perl 5 standard library (5.22.1 as distributed on Ubuntu 16.04)
- The Rust standard library (1.17.0)
- rust-encoding (0.2.33)
- encoding_rs (0.6.10) (of interest to me, because I wrote it; not claiming prominence, yet, since it has not yet been shipped in Firefox ☺)
- Go built-in iteration (1.6.2 as distributed on Ubuntu 16.04)

Why these? Browsers should obviously be considered. I already had copypaste-ready code for ICU, Win32, rust-encoding and the Rust standard library. Java, Ruby, Python 3, Python 2 and Perl 5 were trivial to test due to packaging on Ubuntu and either prior knowledge of how to test or the documentation being approachable. I tested Go after nigeltao on HN pointed out to me what to test. On the other hand, ~~I timed out trying to find the right API entry point in Go documentation, and~~ I timed out trying to get GLib to behave (the glibc layer does not handle REPLACEMENT CHARACTER emission). I figured that testing e.g. CoreFoundation (which I believe only wraps ICU but, who knows, could do something else for UTF-8 like WebKit does), Qt or .Net would have taken too much time *for me*. In any case, the above list should be broad enough to make statements about “multiple prominent implementations”.

I used a specially-crafted HTML document as test input. Since the file is malformed, if you follow the link in your browser, the number of REPLACEMENT CHARACTERs depends on the UTF-8 decoder in your browser.

Most implementations produced bit-identical results matching the old Unicode preference. Therefore, I’m providing only one copy of the output. This file is valid UTF-8, so the number of REPLACEMENT CHARACTERs is encoded in the file and does not depend on your browser. This is the result obtained with (by visual inspection only for browsers):

- Firefox
- Chrome
- Ruby
- Python 3
- The Rust standard library
- rust-encoding

- encoding_rs

~~An interesting browser behavior (link to manual synthesis of valid UTF-8 that looks the way the test input shows up in these browsers) is to emit as many REPLACEMENT CHARACTERs as there are bogus bytes without collapsing sequences that are prefixes of a valid sequence. This is the behavior of:~~

An interesting other multi-implementation behavior is to emit as many REPLACEMENT CHARACTERs as there are bogus bytes without collapsing sequences that are prefixes of a valid sequence (i.e. for a truncated sequence the decoder emits a REPLACEMENT CHARACTER for the lead, resets to the start state and continues immediately after the lead). For Go, this behavior is part of the language spec. This is the result obtained with (by visual inspection only for browsers):

- Edge
- IE
- Safari
- Go

The rest all had mutually different results (links point to valid output created with these implementations):

- Win32 seems to approximate the old guidance with the quirk that for three or four-byte invalid sequences that follow the UTF-8 bit pattern the number of REPLACEMENT CHARACTERs is one fewer than the number of bytes.
- OpenJDK 8 follows the old guidance except it follows the new guidance for CESU-8-encoded surrogates.
- Python 2 (both narrow and wide) exhibits non-conforming behavior by accepting CESU-8-encoded astral code points and round-tripping CESU-8-encoded lone surrogates. Non-shortest two, three or four-byte sequences follow neither the old nor the new guidance.
- Perl 5 follows the old guidance for non-shortest forms but follows the new guidance for CESU-8-encoded surrogates and values above the Unicode range (with knowledge of five and six-byte sequences). Follows the Go behavior of one REPLACEMENT CHARACTER per byte for truncated sequences.
- ICU follows the new guidance, including for five and six-byte sequences for which the proposal and decision were ambiguous.

As you can see, ICU is the most different from the others. In particular, even though Edge, IE11, Safari, Go, OpenJDK 8 and Perl 5 do not follow the old guidance for everything, they match the old guidance for non-shortest forms.

When there are multiple prominent implementations following the old guidance and only ICU following the new guidance if it is taken to include five and six-byte sequences and *no implementation* (that I know of) following the new guidance if it is taken to exclude five and six-byte sequences, I think changing the spec shows poor standard stewardship.

It is wasteful if the implementors who followed the previous advice need to explain why they don't follow the new "best practice". It should be to the developers of the other implementations shouldering the burden of explaining their deviations from the "best practice". It is even more wasteful if the change of Unicode Standard-expressed preference results in code changes in any of the implementations that implemented the old preference, since this would result in implementation and quality assurance work (potentially partially in the form of fixing bugs introduced as part of making changes).

A well-managed standard should not induce, for flimsy reasons, such waste on implementors who trusted the standard previously. Changes to widely-implemented long-standing standards should have a very important and strong rationale. The change at hand lacks such a strong rationale.

Favoring the Unicode-Hosted Implementation

Now that ICU is a Unicode Consortium project, I think the Unicode Consortium should be particularly sensitive to biases arising from being both the source of the spec and the source of a popular implementation. It looks *really bad* both in terms of equal footing of ICU vs. other implementations for the purpose of how the standard is developed as well as the reliability of the standard text vs. ICU source code as the source of truth that other implementors need to pay attention to if the way the Unicode Consortium resolves a discrepancy between ICU behavior and a well-known spec provision (even if mere expression of preference that isn't a conformance requirement) is by changing the spec instead of changing ICU.

The Mere Preference Has Been Elevated into a Requirement Elsewhere

Even though the Unicode Standard expresses the number of REPLACEMENT CHARACTERs as a mere non-normative preference, Web standards these days tend to avoid implementation-defined behavior due to the painful experience of Web sites developing dependencies on the quirks of particular browsers in areas that old specs considered error situations not worth spelling out precise processing requirements for. Therefore, there has been a long push towards well-defined behavior even in error situations on the Web without debating each particular error case individually to assess if the case at hand is prone to sites developing dependencies on a particular behavior. (To be clear, I am not claiming that the number of REPLACEMENT CHARACTERs would be particularly prone to browser-specific site dependences.)

As a result, the WHATWG Encoding Standard, which seeks to be normative over Web browsers, has precise requirements for REPLACEMENT CHARACTER emission. For UTF-8, these used to differ from the preference expressed by the Unicode Standard, but that was reported as a bug in 2012 and the WHATWG Encoding Standard aligned with the preference expressed by the Unicode Standard making it a requirement. Firefox was changed accordingly at the same time.

Chrome changed in 2016 to bring the Web Template Framework part of Chrome into consistency with V8, since a discrepancy between the two caused a bug! The change cited Unicode directly instead of citing the WHATWG Encoding Standard. This Chrome bug is the strongest evidence that I have seen that the precise behavior can actually matter.

Regrettably, the distance to make all browsers do the same thing would have been the shortest before the Chrome change. Before then, the shortest path to making all browsers do the same thing would have been to make V8 and Firefox emit one REPLACEMENT CHARACTER per bogus byte. However, I am not advocating such a change now. The V8 consistency issue shows that UTF-8 decoding comes to browsers from more places in the code than one would expect, some of those are implemented directly downstream of the Unicode Standard instead of downstream of the WHATWG Encoding Standard and consistency between those can turn out to matter. In the case of Firefox, there is the Rust standard library in addition to the main encoding library (currently uconv, encoding_rs hopefully in the near future), and I don't want to ask the Rust standard library to change. (Also, from a purely selfish perspective, replicating the Edge/Safari behavior in encoding_rs, while possible, would lead to added complexity, because it would involve emitting multiple REPLACEMENT CHARACTERs retroactively for bytes that the decoder has already consumed as a valid-looking prefix.)

When two out of four of the major browsers match what the WHATWG Encoding Standard says about UTF-8 decoding and the two others are very close, the WHATWG spec is likely to stay as-is. It's a shame if the Unicode change makes a conformance requirement for the Web differ from the non-requirement preference expressed by the Unicode Standard. There are already enough competing specs and stale spec versions around that making sure that browser developers read the right spec requires constant vigilance. It would be sad to have to add this particular part of the Unicode Standard to the "wrong specs to read" list. Even worse if the Unicode change leads to more bugs like the discrepancy between V8 and WTF within Chrome.

Process Issues

This is mostly of curiosity value but may be of relevance for the purpose of getting the decision overturned. It appears that the agenda for a Unicode Technical Committee meeting is supposed to be set at least a week in advance of the meeting, but the proposal at issue here seems to have been submitted on a shorter notice (proposal dated May 11 and accepted on May 12). Also, the old preference was formulated as the outcome of a more heavy-weight Public Review Issue process, so it seems inappropriate to change the outcome of a heavy-weight process by using a lighter-weight decision process.

Where to Go from Here?

First, I hope that the decision to change the preference that the Unicode Standard expresses for the number of REPLACEMENT CHARACTERs is overturned on appeal for the above reasons and for the bad precedent that the change is suggestive of when viewed as a slippery slope towards changing more important things on flimsy grounds. (Or, alternatively, I hope the Unicode Standard stops expressing a preference for the number of REPLACEMENT CHARACTERs altogether beyond "at least one and no more than the number of bogus bytes".)

Second, I hope that the Unicode Consortium takes steps to mitigate the risk of making decisions on flimsy grounds in the future by requiring proposals to change text concerning implementation behavior (regardless of whether an actual requirement or a mere expression of preference) to come with a survey of the behavior of a large number of prominent existing implementations. The more established a given behavior is in implementations, the stronger the rationale should be to change the required or preferred behavior. The Unicode Consortium-hosted implementation should have no special weight when considering the existing implementation landscape.

That is, I think the proposal to change the preferred behavior in this case should have come with the kind of investigation that I performed here, preferably considering even more implementations, instead of baiting someone other than the person making the proposal to do the investigation after the decision has already been taken.

Henri Sivonen

Text written: 2017-05-31. Updated 2017-06-09 with results for Go and with a typo fix to a legend in the test file (regenerated the result files accordingly). Updated 2017-06-12 with observation about Python 2 generating ill-formed output for lone surrogates.

Main page

A copy of https://hsivonen.fi/broken-utf-8/spec.html, the preferred behavior according to the guidance prior to UTC decision 151-C19 follows. This is the behavior of:

- Firefox
- Chrome
- Ruby
- Python 3
- The Rust standard library
- rust-encoding
- encoding_rs

Broken UTF-8

Five-byte and six-byte sequences were defined in RFC 2297 but are no longer part of the UTF-8 definition.

Non-shortest forms for lowest single-byte (U+0000)

Two-byte sequence (C0 80)

��

Three-byte sequence (E0 80 80)

���

Four-byte sequence (F0 80 80 80)

����

Five-byte sequence (F8 80 80 80 80)

�����

Six-byte sequence (FC 80 80 80 80 80)

    ������

Non-shortest forms for highest single-byte (U+007F)

Two-byte sequence (C1 BF)

    ��

Three-byte sequence (E0 81 BF)

    ���

Four-byte sequence (F0 80 81 BF)

    ����

Five-byte sequence (F8 80 80 81 BF)

    �����

Six-byte sequence (FC 80 80 80 81 BF)

    ������

Non-shortest forms for lowest two-byte (U+0080)

Three-byte sequence (E0 82 80)

    ���

Four-byte sequence (F0 80 82 80)

    ����

Five-byte sequence (F8 80 80 82 80)

    �����

Six-byte sequence (FC 80 80 80 82 80)

    ������

Non-shortest forms for highest two-byte (U+07FF)

Three-byte sequence (E0 9F BF)

    ���

Four-byte sequence (F0 80 9F BF)

    ����

Five-byte sequence (F8 80 80 9F BF)

    �����

Six-byte sequence (FC 80 80 80 9F BF)

    ������

Non-shortest forms for lowest three-byte (U+0800)

Four-byte sequence (F0 80 A0 80)

����

Five-byte sequence (F8 80 80 A0 80)

�����

Six-byte sequence (FC 80 80 80 A0 80)

������

Non-shortest forms for highest three-byte (U+FFFF)

Four-byte sequence (F0 8F BF BF)

����

Five-byte sequence (F8 80 8F BF BF)

�����

Six-byte sequence (FC 80 80 8F BF BF)

������

Non-shortest forms for lowest four-byte (U+10000)

Five-byte sequence (F8 80 90 80 80)

�����

Six-byte sequence (FC 80 80 90 80 80)

������

Non-shortest forms for last Unicode (U+10FFFF)

Five-byte sequence (F8 84 8F BF BF)

�����

Six-byte sequence (FC 80 84 8F BF BF)

������

Out of range

One past Unicode (F4 90 80 80)

����

Longest five-byte sequence (FB BF BF BF BF)

�����

Longest six-byte sequence (FD BF BF BF BF BF)

������

First surrogate (ED A0 80)

￼￼￼

Last surrogate (ED BF BF)

���

CESU-8 surrogate pair (ED A0 BD ED B2 A9)

������

Out of range and non-shortest

One past Unicode as five-byte sequence (F8 84 90 80 80)

�����

One past Unicode as six-byte sequence (FC 80 84 90 80 80)

������

First surrogate as four-byte sequence (F0 8D A0 80)

����

Last surrogate as four-byte sequence (F0 8D BF BF)

����

CESU-8 surrogate pair as two four-byte overlongs (F0 8D A0 BD F0 8D B2 A9)

��������

Lone trails

One (80)

�

Two (80 80)

��

Three (80 80 80)

���

Four (80 80 80 80)

����

Five (80 80 80 80 80)

�����

Six (80 80 80 80 80 80)

������

Seven (80 80 80 80 80 80 80)

�������

After valid two-byte (C2 B6 80)

¶�

After valid three-byte (E2 98 83 80)

☃�

After valid four-byte (F0 9F 92 A9 80)

💩�

After five-byte (FB BF BF BF BF 80)

������

After six-byte (FD BF BF BF BF BF 80)

�������

Truncated sequences

Two-byte lead (C2)

�

Three-byte lead (E2)

�

Three-byte lead and one trail (E2 98)

�

Four-byte lead (F0)

�

Four-byte lead and one trail (F0 9F)

�

Four-byte lead and two trails (F0 9F 92)

�

Leftovers

FE (FE)

�

FE and trail (FE 80)

��

FF (FF)

�

FF and trail (FF 80)

��

A copy of https://hsivonen.fi/broken-utf-8/one-per-byte.html follows. This is the behavior of:

- Edge
- IE
- Safari
- Go

Broken UTF-8

Five-byte and six-byte sequences were defined in RFC 2297 but are no longer part of the UTF-8 definition.

Non-shortest forms for lowest single-byte (U+0000)

Two-byte sequence (C0 80)

��

Three-byte sequence (E0 80 80)

���

Four-byte sequence (F0 80 80 80)

����

Five-byte sequence (F8 80 80 80 80)

�����

Six-byte sequence (FC 80 80 80 80 80)

������

Non-shortest forms for highest single-byte (U+007F)

Two-byte sequence (C1 BF)

��

Three-byte sequence (E0 81 BF)

���

Four-byte sequence (F0 80 81 BF)

����

Five-byte sequence (F8 80 80 81 BF)

�����

Six-byte sequence (FC 80 80 80 81 BF)

������

Non-shortest forms for lowest two-byte (U+0080)

Three-byte sequence (E0 82 80)

���

Four-byte sequence (F0 80 82 80)

����

Five-byte sequence (F8 80 80 82 80)

�����

Six-byte sequence (FC 80 80 80 82 80)

������

Non-shortest forms for highest two-byte (U+07FF)

Three-byte sequence (E0 9F BF)

���

Four-byte sequence (F0 80 9F BF)

����

Five-byte sequence (F8 80 80 9F BF)

�����

Six-byte sequence (FC 80 80 80 9F BF)

������

Non-shortest forms for lowest three-byte (U+0800)

Four-byte sequence (F0 80 A0 80)

����

Five-byte sequence (F8 80 80 A0 80)

�����

Six-byte sequence (FC 80 80 80 A0 80)

������

Non-shortest forms for highest three-byte (U+FFFF)

Four-byte sequence (F0 8F BF BF)

����

Five-byte sequence (F8 80 8F BF BF)

�����

Six-byte sequence (FC 80 80 8F BF BF)

    ������

Non-shortest forms for lowest four-byte (U+10000)

Five-byte sequence (F8 80 90 80 80)

    �����

Six-byte sequence (FC 80 80 90 80 80)

    ������

Non-shortest forms for last Unicode (U+10FFFF)

Five-byte sequence (F8 84 8F BF BF)

    �����

Six-byte sequence (FC 80 84 8F BF BF)

    ������

Out of range

One past Unicode (F4 90 80 80)

    ����

Longest five-byte sequence (FB BF BF BF BF)

    �����

Longest six-byte sequence (FD BF BF BF BF BF)

    ������

First surrogate (ED A0 80)

    ���

Last surrogate (ED BF BF)

    ���

CESU-8 surrogate pair (ED A0 BD ED B2 A9)

    ������

Out of range and non-shortest

One past Unicode as five-byte sequence (F8 84 90 80 80)

    �����

One past Unicode as six-byte sequence (FC 80 84 90 80 80)

    ������

First surrogate as four-byte sequence (F0 8D A0 80)

￼￼￼￼

Last surrogate as four-byte sequence (F0 8D BF BF)

￼￼￼￼

CESU-8 surrogate pair as two four-byte overlongs (F0 8D A0 BD F0 8D B2 A9)

￼￼￼￼￼￼￼￼

Lone trails

One (80)

￼

Two (80 80)

￼￼

Three (80 80 80)

￼￼￼

Four (80 80 80 80)

￼￼￼￼

Five (80 80 80 80 80)

￼￼￼￼￼

Six (80 80 80 80 80 80)

￼￼￼￼￼￼

Seven (80 80 80 80 80 80 80)

￼￼￼￼￼￼￼

After valid two-byte (C2 B6 80)

¶￼

After valid three-byte (E2 98 83 80)

☃￼

After valid four-byte (F0 9F 92 A9 80)

💩￼

After five-byte (FB BF BF BF BF 80)

￼￼￼￼￼￼

After six-byte (FD BF BF BF BF BF 80)

￼￼￼￼￼￼￼

Truncated sequences

Two-byte lead (C2)

￼

Three-byte lead (E2)

�

Three-byte lead and one trail (E2 98)

��

Four-byte lead (F0)

�

Four-byte lead and one trail (F0 9F)

��

Four-byte lead and two trails (F0 9F 92)

���

Leftovers

FE (FE)

�

FE and trail (FE 80)

��

FF (FF)

�

FF and trail (FF 80)

��

A copy of https://hsivonen.fi/broken-utf-8/win32.html follows. This is the behavior of Win32 (`MultiByteToWideChar` on Windows 10 1607).

Broken UTF-8

Five-byte and six-byte sequences were defined in RFC 2297 but are no longer part of the UTF-8 definition.

Non-shortest forms for lowest single-byte (U+0000)

Two-byte sequence (C0 80)

��

Three-byte sequence (E0 80 80)

��

Four-byte sequence (F0 80 80 80)

���

Five-byte sequence (F8 80 80 80 80)

�����

Six-byte sequence (FC 80 80 80 80 80)

������

Non-shortest forms for highest single-byte (U+007F)

Two-byte sequence (C1 BF)

��

Three-byte sequence (E0 81 BF)

��

Four-byte sequence (F0 80 81 BF)

���

Five-byte sequence (F8 80 80 81 BF)

�����

Six-byte sequence (FC 80 80 80 81 BF)

������

Non-shortest forms for lowest two-byte (U+0080)

Three-byte sequence (E0 82 80)

��

Four-byte sequence (F0 80 82 80)

���

Five-byte sequence (F8 80 80 82 80)

�����

Six-byte sequence (FC 80 80 80 82 80)

������

Non-shortest forms for highest two-byte (U+07FF)

Three-byte sequence (E0 9F BF)

��

Four-byte sequence (F0 80 9F BF)

    ���

Five-byte sequence (F8 80 80 9F BF)

    �����

Six-byte sequence (FC 80 80 80 9F BF)

    ������

Non-shortest forms for lowest three-byte (U+0800)

Four-byte sequence (F0 80 A0 80)

    ���

Five-byte sequence (F8 80 80 A0 80)

    �����

Six-byte sequence (FC 80 80 80 A0 80)

    ������

Non-shortest forms for highest three-byte (U+FFFF)

Four-byte sequence (F0 8F BF BF)

    ���

Five-byte sequence (F8 80 8F BF BF)

    �����

Six-byte sequence (FC 80 80 8F BF BF)

    ������

Non-shortest forms for lowest four-byte (U+10000)

Five-byte sequence (F8 80 90 80 80)

    �����

Six-byte sequence (FC 80 80 90 80 80)

    ������

Non-shortest forms for last Unicode (U+10FFFF)

Five-byte sequence (F8 84 8F BF BF)

    �����

Six-byte sequence (FC 80 84 8F BF BF)

    ������

Out of range

One past Unicode (F4 90 80 80)
: ���

Longest five-byte sequence (FB BF BF BF BF)
: �����

Longest six-byte sequence (FD BF BF BF BF BF)
: ������

First surrogate (ED A0 80)
: ��

Last surrogate (ED BF BF)
: ��

CESU-8 surrogate pair (ED A0 BD ED B2 A9)
: ����

Out of range and non-shortest

One past Unicode as five-byte sequence (F8 84 90 80 80)
: �����

One past Unicode as six-byte sequence (FC 80 84 90 80 80)
: ������

First surrogate as four-byte sequence (F0 8D A0 80)
: ���

Last surrogate as four-byte sequence (F0 8D BF BF)
: ���

CESU-8 surrogate pair as two four-byte overlongs (F0 8D A0 BD F0 8D B2 A9)
: ������

Lone trails

One (80)
: �

Two (80 80)
: ��

Three (80 80 80)
: ���

Four (80 80 80 80)
: ����

Five (80 80 80 80 80)

�����

Six (80 80 80 80 80 80)

������

Seven (80 80 80 80 80 80 80)

�������

After valid two-byte (C2 B6 80)

¶�

After valid three-byte (E2 98 83 80)

☃�

After valid four-byte (F0 9F 92 A9 80)

💩�

After five-byte (FB BF BF BF BF 80)

������

After six-byte (FD BF BF BF BF BF 80)

�������

Truncated sequences

Two-byte lead (C2)

�

Three-byte lead (E2)

�

Three-byte lead and one trail (E2 98)

�

Four-byte lead (F0)

�

Four-byte lead and one trail (F0 9F)

�

Four-byte lead and two trails (F0 9F 92)

�

Leftovers

FE (FE)

�

FE and trail (FE 80)

��

FF (FF)
: �

FF and trail (FF 80)
: ��

A copy of https://hsivonen.fi/broken-utf-8/java.html follows. This is the behavior of OpenJDK 8.

Broken UTF-8

Five-byte and six-byte sequences were defined in RFC 2297 but are no longer part of the UTF-8 definition.

Non-shortest forms for lowest single-byte (U+0000)

Two-byte sequence (C0 80)
: ��

Three-byte sequence (E0 80 80)
: ���

Four-byte sequence (F0 80 80 80)
: ����

Five-byte sequence (F8 80 80 80 80)
: �����

Six-byte sequence (FC 80 80 80 80 80)
: ������

Non-shortest forms for highest single-byte (U+007F)

Two-byte sequence (C1 BF)
: ��

Three-byte sequence (E0 81 BF)
: ���

Four-byte sequence (F0 80 81 BF)
: ����

Five-byte sequence (F8 80 80 81 BF)

�����

Six-byte sequence (FC 80 80 80 81 BF)

������

Non-shortest forms for lowest two-byte (U+0080)

Three-byte sequence (E0 82 80)

���

Four-byte sequence (F0 80 82 80)

����

Five-byte sequence (F8 80 80 82 80)

�����

Six-byte sequence (FC 80 80 80 82 80)

������

Non-shortest forms for highest two-byte (U+07FF)

Three-byte sequence (E0 9F BF)

���

Four-byte sequence (F0 80 9F BF)

����

Five-byte sequence (F8 80 80 9F BF)

�����

Six-byte sequence (FC 80 80 80 9F BF)

������

Non-shortest forms for lowest three-byte (U+0800)

Four-byte sequence (F0 80 A0 80)

����

Five-byte sequence (F8 80 80 A0 80)

�����

Six-byte sequence (FC 80 80 80 A0 80)

������

Non-shortest forms for highest three-byte (U+FFFF)

Four-byte sequence (F0 8F BF BF)

> ����

Five-byte sequence (F8 80 8F BF BF)

> �����

Six-byte sequence (FC 80 80 8F BF BF)

> ������

Non-shortest forms for lowest four-byte (U+10000)

Five-byte sequence (F8 80 90 80 80)

> �����

Six-byte sequence (FC 80 80 90 80 80)

> ������

Non-shortest forms for last Unicode (U+10FFFF)

Five-byte sequence (F8 84 8F BF BF)

> �����

Six-byte sequence (FC 80 84 8F BF BF)

> ������

Out of range

One past Unicode (F4 90 80 80)

> ����

Longest five-byte sequence (FB BF BF BF BF)

> �����

Longest six-byte sequence (FD BF BF BF BF BF)

> ������

First surrogate (ED A0 80)

> �

Last surrogate (ED BF BF)

> �

CESU-8 surrogate pair (ED A0 BD ED B2 A9)

> ��

Out of range and non-shortest

One past Unicode as five-byte sequence (F8 84 90 80 80)

> �����

One past Unicode as six-byte sequence (FC 80 84 90 80 80)

  ������

First surrogate as four-byte sequence (F0 8D A0 80)

  ����

Last surrogate as four-byte sequence (F0 8D BF BF)

  ����

CESU-8 surrogate pair as two four-byte overlongs (F0 8D A0 BD F0 8D B2 A9)

  ��������

Lone trails

One (80)

  �

Two (80 80)

  ��

Three (80 80 80)

  ���

Four (80 80 80 80)

  ����

Five (80 80 80 80 80)

  �����

Six (80 80 80 80 80 80)

  ������

Seven (80 80 80 80 80 80 80)

  �������

After valid two-byte (C2 B6 80)

  ¶�

After valid three-byte (E2 98 83 80)

  ☃�

After valid four-byte (F0 9F 92 A9 80)

  💩�

After five-byte (FB BF BF BF BF 80)

  ������

After six-byte (FD BF BF BF BF BF 80)

  �������

Truncated sequences

Two-byte lead (C2)

�

Three-byte lead (E2)

�

Three-byte lead and one trail (E2 98)

�

Four-byte lead (F0)

�

Four-byte lead and one trail (F0 9F)

�

Four-byte lead and two trails (F0 9F 92)

�

Leftovers

FE (FE)

�

FE and trail (FE 80)

��

FF (FF)

�

FF and trail (FF 80)

��

A copy of https://hsivonen.fi/broken-utf-8/python2.html follows with ill-formed output replaced with annotation in italic. This is the behavior of Python 2.7.x (both narrow and wide builds).

Broken UTF-8

Five-byte and six-byte sequences were defined in RFC 2297 but are no longer part of the UTF-8 definition.

Non-shortest forms for lowest single-byte (U+0000)

Two-byte sequence (C0 80)

��

Three-byte sequence (E0 80 80)

��

Four-byte sequence (F0 80 80 80)

��

Five-byte sequence (F8 80 80 80 80)

�����

Six-byte sequence (FC 80 80 80 80 80)

������

Non-shortest forms for highest single-byte (U+007F)

Two-byte sequence (C1 BF)

��

Three-byte sequence (E0 81 BF)

��

Four-byte sequence (F0 80 81 BF)

��

Five-byte sequence (F8 80 80 81 BF)

�����

Six-byte sequence (FC 80 80 80 81 BF)

������

Non-shortest forms for lowest two-byte (U+0080)

Three-byte sequence (E0 82 80)

��

Four-byte sequence (F0 80 82 80)

��

Five-byte sequence (F8 80 80 82 80)

�����

Six-byte sequence (FC 80 80 80 82 80)

������

Non-shortest forms for highest two-byte (U+07FF)

Three-byte sequence (E0 9F BF)

��

Four-byte sequence (F0 80 9F BF)

��

Five-byte sequence (F8 80 80 9F BF)

�����

Six-byte sequence (FC 80 80 80 9F BF)

������

Non-shortest forms for lowest three-byte (U+0800)

Four-byte sequence (F0 80 A0 80)

��

Five-byte sequence (F8 80 80 A0 80)

�����

Six-byte sequence (FC 80 80 80 A0 80)

������

Non-shortest forms for highest three-byte (U+FFFF)

Four-byte sequence (F0 8F BF BF)

��

Five-byte sequence (F8 80 8F BF BF)

�����

Six-byte sequence (FC 80 80 8F BF BF)

������

Non-shortest forms for lowest four-byte (U+10000)

Five-byte sequence (F8 80 90 80 80)

�����

Six-byte sequence (FC 80 80 90 80 80)

������

Non-shortest forms for last Unicode (U+10FFFF)

Five-byte sequence (F8 84 8F BF BF)

�����

Six-byte sequence (FC 80 84 8F BF BF)

    ������

Out of range

One past Unicode (F4 90 80 80)

    ��

Longest five-byte sequence (FB BF BF BF BF)

    �����

Longest six-byte sequence (FD BF BF BF BF BF)

    ������

First surrogate (ED A0 80)

    The bytes ED A0 80

Last surrogate (ED BF BF)

    The bytes ED BF BF

CESU-8 surrogate pair (ED A0 BD ED B2 A9)

    💩

Out of range and non-shortest

One past Unicode as five-byte sequence (F8 84 90 80 80)

    �����

One past Unicode as six-byte sequence (FC 80 84 90 80 80)

    ������

First surrogate as four-byte sequence (F0 8D A0 80)

    ��

Last surrogate as four-byte sequence (F0 8D BF BF)

    ��

CESU-8 surrogate pair as two four-byte overlongs (F0 8D A0 BD F0 8D B2 A9)

    ����

Lone trails

One (80)

    �

Two (80 80)

    ��

Three (80 80 80)

　���

Four (80 80 80 80)

　����

Five (80 80 80 80 80)

　�����

Six (80 80 80 80 80 80)

　������

Seven (80 80 80 80 80 80 80)

　�������

After valid two-byte (C2 B6 80)

　¶�

After valid three-byte (E2 98 83 80)

　☃�

After valid four-byte (F0 9F 92 A9 80)

　💩�

After five-byte (FB BF BF BF BF 80)

　������

After six-byte (FD BF BF BF BF BF 80)

　�������

Truncated sequences

Two-byte lead (C2)

　�

Three-byte lead (E2)

　�

Three-byte lead and one trail (E2 98)

　�

Four-byte lead (F0)

　�

Four-byte lead and one trail (F0 9F)

　�

Four-byte lead and two trails (F0 9F 92)

　�

Leftovers

FE (FE)
    �

FE and trail (FE 80)
    ��

FF (FF)
    �

FF and trail (FF 80)
    ��

A copy of https://hsivonen.fi/broken-utf-8/perl5.html follows. This is the behavior of Perl 5.

Broken UTF-8

Five-byte and six-byte sequences were defined in RFC 2297 but are no longer part of the UTF-8 definition.

Non-shortest forms for lowest single-byte (U+0000)

Two-byte sequence (C0 80)
    ��

Three-byte sequence (E0 80 80)
    ���

Four-byte sequence (F0 80 80 80)
    ����

Five-byte sequence (F8 80 80 80 80)
    �����

Six-byte sequence (FC 80 80 80 80 80)
    ������

Non-shortest forms for highest single-byte (U+007F)

Two-byte sequence (C1 BF)
    ��

Three-byte sequence (E0 81 BF)

���

Four-byte sequence (F0 80 81 BF)

����

Five-byte sequence (F8 80 80 81 BF)

�����

Six-byte sequence (FC 80 80 80 81 BF)

������

Non-shortest forms for lowest two-byte (U+0080)

Three-byte sequence (E0 82 80)

���

Four-byte sequence (F0 80 82 80)

����

Five-byte sequence (F8 80 80 82 80)

�����

Six-byte sequence (FC 80 80 80 82 80)

������

Non-shortest forms for highest two-byte (U+07FF)

Three-byte sequence (E0 9F BF)

���

Four-byte sequence (F0 80 9F BF)

����

Five-byte sequence (F8 80 80 9F BF)

�����

Six-byte sequence (FC 80 80 80 9F BF)

������

Non-shortest forms for lowest three-byte (U+0800)

Four-byte sequence (F0 80 A0 80)

����

Five-byte sequence (F8 80 80 A0 80)

�����

Six-byte sequence (FC 80 80 80 A0 80)

������

Non-shortest forms for highest three-byte (U+FFFF)

Four-byte sequence (F0 8F BF BF)

����

Five-byte sequence (F8 80 8F BF BF)

�����

Six-byte sequence (FC 80 80 8F BF BF)

������

Non-shortest forms for lowest four-byte (U+10000)

Five-byte sequence (F8 80 90 80 80)

�����

Six-byte sequence (FC 80 80 90 80 80)

������

Non-shortest forms for last Unicode (U+10FFFF)

Five-byte sequence (F8 84 8F BF BF)

�����

Six-byte sequence (FC 80 84 8F BF BF)

������

Out of range

One past Unicode (F4 90 80 80)

�

Longest five-byte sequence (FB BF BF BF BF)

�

Longest six-byte sequence (FD BF BF BF BF BF)

�

First surrogate (ED A0 80)

�

Last surrogate (ED BF BF)

�

CESU-8 surrogate pair (ED A0 BD ED B2 A9)

��

Out of range and non-shortest

One past Unicode as five-byte sequence (F8 84 90 80 80)

�����

One past Unicode as six-byte sequence (FC 80 84 90 80 80)

������

First surrogate as four-byte sequence (F0 8D A0 80)

����

Last surrogate as four-byte sequence (F0 8D BF BF)

����

CESU-8 surrogate pair as two four-byte overlongs (F0 8D A0 BD F0 8D B2 A9)

��������

Lone trails

One (80)

�

Two (80 80)

��

Three (80 80 80)

���

Four (80 80 80 80)

����

Five (80 80 80 80 80)

�����

Six (80 80 80 80 80 80)

������

Seven (80 80 80 80 80 80 80)

�������

After valid two-byte (C2 B6 80)

¶�

After valid three-byte (E2 98 83 80)

☃�

After valid four-byte (F0 9F 92 A9 80)

💩�

After five-byte (FB BF BF BF BF 80)

��

After six-byte (FD BF BF BF BF BF 80)

��

Truncated sequences

Two-byte lead (C2)

�

Three-byte lead (E2)

�

Three-byte lead and one trail (E2 98)

��

Four-byte lead (F0)

�

Four-byte lead and one trail (F0 9F)

��

Four-byte lead and two trails (F0 9F 92)

���

Leftovers

FE (FE)

�

FE and trail (FE 80)

��

FF (FF)

�

FF and trail (FF 80)

��

A copy of https://hsivonen.fi/broken-utf-8/icu.html follows. This is the behavior of ICU.

Broken UTF-8

Five-byte and six-byte sequences were defined in RFC 2297 but are no longer part of the UTF-8 definition.

Non-shortest forms for lowest single-byte (U+0000)

Two-byte sequence (C0 80)

 �

Three-byte sequence (E0 80 80)

 �

Four-byte sequence (F0 80 80 80)

 �

Five-byte sequence (F8 80 80 80 80)

 �

Six-byte sequence (FC 80 80 80 80 80)

 �

Non-shortest forms for highest single-byte (U+007F)

Two-byte sequence (C1 BF)

 �

Three-byte sequence (E0 81 BF)

 �

Four-byte sequence (F0 80 81 BF)

 �

Five-byte sequence (F8 80 80 81 BF)

 �

Six-byte sequence (FC 80 80 80 81 BF)

 �

Non-shortest forms for lowest two-byte (U+0080)

Three-byte sequence (E0 82 80)

 �

Four-byte sequence (F0 80 82 80)

 �

Five-byte sequence (F8 80 80 82 80)

 �

Six-byte sequence (FC 80 80 80 82 80)

�

Non-shortest forms for highest two-byte (U+07FF)

Three-byte sequence (E0 9F BF)

�

Four-byte sequence (F0 80 9F BF)

�

Five-byte sequence (F8 80 80 9F BF)

�

Six-byte sequence (FC 80 80 80 9F BF)

�

Non-shortest forms for lowest three-byte (U+0800)

Four-byte sequence (F0 80 A0 80)

�

Five-byte sequence (F8 80 80 A0 80)

�

Six-byte sequence (FC 80 80 80 A0 80)

�

Non-shortest forms for highest three-byte (U+FFFF)

Four-byte sequence (F0 8F BF BF)

�

Five-byte sequence (F8 80 8F BF BF)

�

Six-byte sequence (FC 80 80 8F BF BF)

�

Non-shortest forms for lowest four-byte (U+10000)

Five-byte sequence (F8 80 90 80 80)

�

Six-byte sequence (FC 80 80 90 80 80)

�

Non-shortest forms for last Unicode (U+10FFFF)

Five-byte sequence (F8 84 8F BF BF)

  �

Six-byte sequence (FC 80 84 8F BF BF)

  �

Out of range

One past Unicode (F4 90 80 80)

  �

Longest five-byte sequence (FB BF BF BF BF)

  �

Longest six-byte sequence (FD BF BF BF BF BF)

  �

First surrogate (ED A0 80)

  �

Last surrogate (ED BF BF)

  �

CESU-8 surrogate pair (ED A0 BD ED B2 A9)

  ��

Out of range and non-shortest

One past Unicode as five-byte sequence (F8 84 90 80 80)

  �

One past Unicode as six-byte sequence (FC 80 84 90 80 80)

  �

First surrogate as four-byte sequence (F0 8D A0 80)

  �

Last surrogate as four-byte sequence (F0 8D BF BF)

  �

CESU-8 surrogate pair as two four-byte overlongs (F0 8D A0 BD F0 8D B2 A9)

  ��

Lone trails

One (80)

  �

Two (80 80)

��

Three (80 80 80)

���

Four (80 80 80 80)

����

Five (80 80 80 80 80)

�����

Six (80 80 80 80 80 80)

������

Seven (80 80 80 80 80 80 80)

�������

After valid two-byte (C2 B6 80)

¶�

After valid three-byte (E2 98 83 80)

☃�

After valid four-byte (F0 9F 92 A9 80)

💩�

After five-byte (FB BF BF BF BF 80)

��

After six-byte (FD BF BF BF BF BF 80)

��

Truncated sequences

Two-byte lead (C2)

�

Three-byte lead (E2)

�

Three-byte lead and one trail (E2 98)

�

Four-byte lead (F0)

�

Four-byte lead and one trail (F0 9F)

�

Four-byte lead and two trails (F0 9F 92)

�

Leftovers

FE (FE)

�

FE and trail (FE 80)

��

FF (FF)

�

FF and trail (FF 80)

��

(End of submission)