L2/19-192

Review of Unicode 2018 Henri Sivonen docs

Markus Scherer 2019-may-01

Security considerations

https://www.unicode.org/L2/L2018/18185-security.pdf

156-A16	Markus Scherer	Review L2/18-185 and make any recommendations back to the UTC.	L2/18-185

Character Encoding Forms C10

Unicode12.0.0/ch03.pdf 3.2 Conformance Requirements / Character Encoding Forms / C10 on p. 83:

> *When a process interprets a code unit sequence which purports to be in a Unicode character encoding form, it shall treat ill-formed code unit sequences as an error condition and shall not interpret such sequences as characters.*
>
> - For example, in UTF-8 every code unit of the form $110xxxx_2$ *must* be followed by a code unit of the form $10xxxxxx_2$. A sequence such as $110xxxxx_2$ $0xxxxxxx_2$ is ill-formed and must never be generated. When faced with this ill-formed code unit sequence while transforming or interpreting text, a conformant process must treat the first code unit $110xxxxx_2$ as an illegally terminated code unit sequence—for example, by signaling an error, filtering the code unit out, or representing the code unit with a marker such as U+FFFD replacement character.

Henri writes:

> Just filtering the code unit out is more dangerous than the two alternatives presented [...] For example, in the context of a data format such as HTML that can carry both inactive content (human-readable text) and active content (JavaScript program code), it is important that errors are not simply removed such that text before and after the error joins together to form code with a potentially dangerous function.
>
> I suggest striking ", filtering the code unit out," from the example and adding another sentence at the end of the paragraph saying: "While simply ignoring an illformed code unit sequence qualifies as not interpreting it as characters, silently ignoring ill-formed sequences is ill-advised, because joining text from before the illformed sequence and after the ill-formed sequence can cause the resulting text to take a new meaning, which might be especially dangerous in the context of textual formats that carry embedded program code, such as JavaScript."

Markus: Looks ok to me. Recommend sending this to the editorial committee.

Reserved & private use

Unicode12.0.0/ch05.pdf 5.3 Unknown and Missing Characters on p. 201:

> ***Reserved and Private-Use Character Codes.*** There are two classes of code points that even a "complete" implementation of the Unicode Standard cannot necessarily interpret correctly:
>
> - Code points that are reserved
> - Code points in the Private Use Area for which no private agreement exists
>
> An implementation should not attempt to interpret such code points. However, in practice, applications must deal with unassigned code points or private-use characters. This may occur, for example, when the application is handling text that originated on a system implementing a later release of the Unicode Standard, with additional assigned characters.
>
> Options for rendering such unknown code points include printing the code point as four to six hexadecimal digits, printing a black or white box, using appropriate glyphs such as [some glyph] for reserved and [some other glyph] for private use, or simply displaying nothing. An implementation should not blindly delete such characters, nor should it unintentionally transform them into something else.

Henri writes:

> ... paragraph says that "simply display nothing" is an option for rendering unknown code points. While the same page goes on to discuss the concept of Default Ignorable Code Points, it would be prudent to qualify the "simply display nothing" option by scoping it to default ignorables in order not to suggest that display nothing is a generally valid fallback (as displaying nothing can mislead the human reader).

Markus: Looks ok to me, although less of a security issue than deleting the code points. I would add as an option displaying the same glyph as for U+FFFD. Recommend sending this to the editorial committee.

Unicode12.0.0/ch05.pdf 5.22 U+FFFD Substitution in Conversion on p. 254 (= last page of chapter 5):

> For conversion *between* different encoding forms of the Unicode Standard, “U+FFFD Substitution of Maximal Subparts” in *Section 3.9, Unicode Encoding Forms* defines a practice for the use of U+FFFD which is consistent with the W3C standard for encoding. It is useful to apply the same practice to the conversion from non-Unicode encodings to an encoding form of the Unicode Standard.
>
> This practice is more secure because it does not result in the conversion consuming parts of valid sequences as though they were invalid. It also guarantees at least one replacement character will occur for each instance of an invalid sequence in the original text. Furthermore, this practice can be defined consistently for better interoperability between different implementations of conversion.
>
> For full consistency, it is important for conversion implementations to agree on 1) the exact set of well-formed sequences for the source encoding, 2) all of the mappings for valid sequences, and 3) the details of the practice for handling ill-formed sequences.

Henri refers to Unicode 10 which had an older recommendation.

Henri writes:

> It would be useful to point to the WHATWG Encoding Standard, which defines precise algorithms for Web-relevant legacy encodings, and to articulate the principle that the algorithms in the WHATWG Encoding Standard embody.

Markus: Sections 3.9 and 5.22 do refer to the “W3C standard for encoding” (with a URL in 3.9) since Unicode 11. I recommend doing nothing further.

Henri writes:

I suggest adding text along these lines: “For Web-relevant legacy encodings, the rules for U+FFFD substitution are defined in the WHATWG Encoding Standard. The general principle for ASCII-compatible multi-byte encodings is that when an ASCII-range trail byte occurs in a multi-byte sequence that either doesn’t fit the general byte pattern of the encoding or that fits the pattern but doesn’t identify a mapped character according to the encoding’s mapping table, the trail byte in the ASCII range must not be subsumed into the U+FFFD substitution together with the non-ASCII lead byte but instead must be reprocessed as a single-byte (ASCII) sequence. This ensures that the introduction of an erroneous lead byte cannot mask and ASCII byte which might be security-relevant in a computer-readable syntax such as HTML or JavaScript.”

Markus:

This would be bad practice that would damage well-formed text and unnecessarily *insert* ASCII syntax characters. In many encodings, in particular those for East Asian languages, it is common to set aside ranges of well-formed sequences for Vendor-Defined Characters (VDC), similar to Unicode’s PUA. These are used by publishers for unusual Kanji that their customers need, by Japanese vendors to add emoji, etc.

In the case of emoji, when encoded as Shift-JIS VDC, many of them have a trail byte in the ASCII range. A normal converter that does not have a mapping for such an emoji VDC would map the well-formed two-byte sequence to one U+FFFD. With the proposed practice, it would map the lead byte to U+FFFD and emit an ASCII character like the at sign or curly braces for the trail byte. Turning commonly occurring, well-formed text into garbled text with artificially inserted ASCII characters, just due to missing mappings for some sequences, is a bad idea.

If this were also applied to the DBCS portion of Japanese carrier ISO-2022-JP with emoji VDCs, it would desynchronize the double-byte stream and garble all following text up to the next state change.

I strongly recommend not adopting this suggestion.

Characterization of UTF-32

https://www.unicode.org/L2/L2018/18187-utf-32-fdbk.pdf

156-A19	Markus Scherer	Review L2/18-187 and make any recommendations for updates to chapters 2 and 3 back to the UTC.	L2/18-187

Unicode12.0.0/ch02.pdf 2.5 Encoding Forms on pp. 35-36

UTF-32
...
Preferred Usage. UTF-32 may be a preferred encoding form where memory or disk storage space for characters is not a particular concern, but where fixed-width, single code unit access to characters is desired. UTF-32 is also a preferred encoding form for processing characters on most Unix platforms.

Henri:

Henri points out that some processing needs to look at grapheme clusters rather than just single characters = code points, and he writes that “Unix platforms tend to prefer UTF-8 both on disk and in system APIs”.

He suggests rewriting this paragraph, limiting preferred usage of UTF-32 to single-code point APIs and discounting the use of UTF-32 in wchar_t functions.

Markus:

- I agree that this paragraph would benefit from a revision. UTF-8 has increased in popularity, on Unix and elsewhere.
- I agree with discounting wchar_t. A more modern prominent example of UTF-32 string processing is Python 3. (Logically UTF-32, while storage is optimized.) Also, even when code is generally using UTF-8 or UTF-16, sometimes it will temporarily convert to UTF-32 for easier processing.
- The point about grapheme clusters is valid, but it is still also true that processing in UTF-32 is simpler than in any of the other forms.
- I disagree about “UTF-32 is commonly used for APIs that deal with single code points” because it is misleading to mention any *string* encoding form for *code point* APIs. (They just take code point integers; no UTF involved.)
- Recommend sending this to the editorial committee.

Henri:

As for figure 2-11, it is misleading for the three rows for UTF-32, UTF-16 and UTF-8 to have equal width. It would illustrate the encoding forms better if a byte was allocated equal width on each line so that the line for UTF-32 would be wider than the lines for UTF-16 and UTF-8 (as seen in Figure 2-12 when illustrating encoding schemes). It would probably be the best to keep the lines right-aligned to illustrate that the astral character is equally wide in all three encoding forms. (Right-aligning the rows would improve Figure 2-12 as well.)

Markus:

It looks to me like figure 2-11 wants to focus on code points vs. code units in each of the three encoding forms, and not also compare the relative storage sizes. However, the figure could reasonably be changed to do so. Recommend sending this to the editorial committee.

Markus:

In addition, it may be useful to add one combining mark to the text in figure 2-11 (such as a U+0308 after the A), and/or replacing the cuneiform character with an Emoji sequence, together with adding some text about grapheme clusters (user-perceived “characters” encoded with more than single code points). Recommend sending this to the editorial committee.

Markus:

One other problem I see with the discussion of all three encoding forms in chapter 2.5 is that they are each described with something like “UTF-32 is restricted to representation of code points in the range 0..10FFFF$_{16}$—that is, the Unicode codespace. This guarantees interoperability with the UTF-16 and UTF-8 encoding forms.”

Without reading other parts of the Unicode Standard, this seems to say that surrogates U+D800..U+DFFF are representable in the UTFs, but this has not been true for many years. Formally speaking, the UTFs are restricted to representation of scalar values 0..D7FF and E000..10FFFF. I think we should clarify this.

Recommend sending this to the editorial committee.