Core spec changes for 2018 Henri Sivonen docs

Markus Scherer 2019-oct-10

159-A 120	Markus Scherer	Create a document highlighting proposed changes to the core spec based on document L2/19-192.

See https://www.unicode.org/L2/L2019/19192-review-docs.pdf “Review of Unicode 2018 Henri Sivonen docs”

The following are my suggestions for changes to the core spec, for discussion, and further refinement by the editorial committee.

Security considerations

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

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. While simply ignoring an ill-formed code unit sequence qualifies as not interpreting it as characters, silently ignoring ill-formed sequences is strongly discouraged, because joining text from before the ill-formed sequence and after the ill-formed sequence can cause the resulting text to take a new meaning, which is especially dangerous in the context of textual formats that carry embedded program code, such as JavaScript.

Modifications suggested by Henri with minor spelling changes.

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, displaying the same glyph as for U+FFFD, or using appropriate glyphs such as [some glyph] for reserved and [some other glyph] for private use~~, or simply displaying nothing~~. For certain code points, it is common to simply displaying nothing; see the section on Default Ignorable Code Points below for details. An implementation should not blindly delete such characters, nor should it unintentionally transform them into something else.

Limited "display nothing" option, added U+FFFD glyph option.

Characterization of UTF-32

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

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

UTF-32
UTF-32 is the simplest Unicode encoding form. Each Unicode code point is represented directly by a single 32-bit code unit. Because of this, UTF-32 has a one-to-one relationship between encoded character and code unit; it is a fixed-width character encoding form. ~~This makes UTF-32 an ideal form for APIs that pass single character values.~~

Even when code is generally using UTF-8 or UTF-16, sometimes it will temporarily convert to UTF-32 for easier processing.

Note that, in some use cases, multi-code point sequences are useful or necessary as units of processing; for example, grapheme clusters, or sequences of characters with non-zero combining classes. This may limit the usefulness of a per-code point fixed-width encoding.

Note also that string encoding forms like UTF-32 or UTF-8 are irrelevant for APIs that pass single character values: These typically take or return simple code point integers.

As for all of the Unicode encoding forms, UTF-32 is restricted to representation of code points in the ranges 0..$D7FF_{16}$ and $E000_{16}$..$10FFFF_{16}$—that is, ~~the Unicode codespace~~ Unicode scalar values. This guarantees interoperability with the UTF-16 and UTF-8 encoding forms.

Fixed Width. ...

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.~~ For example, Python 3 strings are sequences of Unicode code points.

Suggested changes inspired by Henri, modified along the lines of the discussion in L2/19-192.

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.