Chapter 2: Remove Comparison of the Advantages of UTFs

Markus Scherer 2019-oct-30

Chapter 2 of the Unicode Standard, at the end of section 2.5 Encoding Forms, includes a subsection “Comparison of the Advantages of UTF-32, UTF-16, and UTF-8”.

I propose that we remove this subsection.

Rationale

This text mostly and unnecessarily repeats advantages of UTF-16 and UTF-32 that are already included in the preceding subsections about each of the UTFs.

This text seems also intended to inform implementation choices, particularly in favor of UTF-16. While I sympathize with that, and while this was useful some years ago, this is no longer useful. Which UTF to use is decided by operating systems, programming languages, libraries, and protocols. Also, some implementations use yet other strategies, such as logically operating in UTF-32 but internally storing Latin-1 bytes if there are no other code points.

Details

I propose that we make the following edits, moving a small amount of text into the UTF-8 subsection and then deleting the Comparison subsection.

The following text is the current Unicode 13 draft, which is slightly modified from https://www.unicode.org/versions/Unicode12.0.0/ch02.pdf pp.36..39.

Note that I recreated enough of the PDF formatting for context (e.g., headings & bullets) but not all styles (e.g., line breaks & some italics).

Proposed additions are underlined and highlighted in green.
Proposed deletions are ~~struck through and highlighted in red~~.
[Editorial comments in brackets.]

UTF-8

To meet the requirements of byte-oriented, ASCII-based systems, a third encoding form is specified by the Unicode Standard: UTF-8. This variable-width encoding form preserves ASCII transparency by making use of 8-bit code units.

Byte-Oriented. Much existing software and practice in information technology have long

depended on character data being represented as a sequence of bytes. Furthermore, many of the protocols depend not only on ASCII values being invariant, but must make use of or avoid special byte values that may have associated control functions. The easiest way to adapt Unicode implementations to such a situation is to make use of an encoding form that is already defined in terms of 8-bit code units and that represents all Unicode characters while not disturbing or reusing any ASCII or C0 control code value. That is the function of UTF-8.

Variable Width. UTF-8 is a variable-width encoding form, using 8-bit code units, in which the high bits of each code unit indicate the part of the code unit sequence to which each byte belongs. A range of 8-bit code unit values is reserved for the first, or leading, element of a UTF-8 code unit sequences, and a completely disjunct range of 8-bit code unit values is reserved for the subsequent, or trailing, elements of such sequences; this convention preserves non-overlap for UTF-8. Table 3-6 on page 126 shows how the bits in a Unicode code point are distributed among the bytes in the UTF-8 encoding form. See Section 3.9, Unicode Encoding Forms, for the full, formal definition of UTF-8.

ASCII Transparency. The UTF-8 encoding form maintains transparency for all of the ASCII code points (0x00..0x7F). That means Unicode code points U+0000..U+007F are converted to single bytes 0x00..0x7F in UTF-8 and are thus indistinguishable from ASCII itself. Furthermore, the values 0x00..0x7F do not appear in any byte for the representation of any other Unicode code point, so that there can be no ambiguity. Beyond the ASCII range of Unicode, many of the non-ideographic scripts are represented by two bytes per code point in UTF-8; all non-surrogate code points between U+0800 and U+FFFF are represented by three bytes; and supplementary code points above U+FFFF require four bytes.

[moved up from Comparison/UTF-8 with changes (& feedback from editors); unsure about best label]
Memory. UTF-8 is reasonably compact in terms of the number of bytes used. Compared with UTF-16, it is much smaller for ASCII syntax and Western languages, but significantly larger for Asian writing systems such as for Hindi, Thai, Chinese, Japanese, and Korean.

Preferred Usage. UTF-8 is typically the preferred encoding form for HTML and similar protocols, particularly for the Internet. The ASCII transparency helps migration. UTF-8 also has the advantage that it is already inherently byte-serialized, as for most existing 8-bit character sets; strings of UTF-8 work easily with the C standard library, and many existing APIs that work for typical East Asian multibyte character sets adapt to UTF-8 as well with little or no change required.

Self-synchronizing. In environments where 8-bit character processing is required for one reason or another, UTF-8 has the following attractive features as compared to other multibyte encodings:

- The first byte of a UTF-8 code unit sequence indicates the number of bytes to follow in a multibyte sequence. This allows for very efficient forward parsing.
- It is efficient to find the start of a character when beginning from an arbitrary location in a byte stream of UTF-8. Programs need to search at most four bytes backward, and usually much less. It is a simple task to recognize an initial byte, because initial bytes are constrained to a fixed range of values.
- As with the other encoding forms, there is no overlap of byte values.

[moved up from Comparison/Binary Sorting without change]

Binary Sorting. A binary sort of UTF-8 strings gives the same ordering as a binary sort of Unicode code points. This is obviously the same order as for a binary sort of UTF-32 strings.

[otherwise delete the whole final subsection]

~~Comparison of the Advantages of UTF-32, UTF-16, and UTF-8~~

~~On the face of it, UTF-32 would seem to be the obvious choice of Unicode encoding forms for an internal processing code because it is a fixed-width encoding form. However, UTF16 has many countervailing advantages that may lead implementers to choose it instead as an internal processing code.~~

~~While all three encoding forms need at most 4 bytes (or 32 bits) of data for each character, in practice UTF-32 in almost all cases for real data sets occupies twice the storage that UTF-16 requires. Therefore, a common strategy is to have internal string storage use UTF16 or UTF-8 but to use UTF-32 when manipulating individual characters.~~

~~UTF-32 Versus UTF-16.~~ ~~On average, more than 99% of all UTF-16 data is expressed using single code units. This includes nearly all of the typical characters that software needs to handle with special operations on text—for example, commonly used format control characters. As a consequence, most text scanning operations do not need to decode UTF-16 surrogate pairs at all, but rather can safely treat them as an opaque part of a character string.~~

~~For many operations, UTF-16 is as easy to handle as UTF-32, and the performance of UTF-16 as a processing code tends to be quite good. UTF-16 is the internal processing code of choice for a majority of implementations supporting Unicode. Other than for Unix platforms, UTF-16 provides the right mix of compact size with the ability to handle the occasional character outside the BMP.~~

~~UTF-32 has somewhat of an advantage when it comes to simplicity of software coding design and maintenance. Because the character handling is fixed width, UTF-32 processing does not require maintaining branches in the software to test and process the double code unit elements required for supplementary characters by UTF-16. Conversely, 32-bit indices into large tables are not particularly memory efficient. To avoid the large memory penalties of such indices, Unicode tables are often handled as multistage tables (see "Multistage Tables" in Section 5.1, Data Structures for Character Conversion). In such cases, the 32-bit code point values are sliced into smaller ranges to permit segmented access to the tables. This is true even in typical UTF-32 implementations.~~

~~The performance of UTF-32 as a processing code may actually be worse than the performance of UTF-16 for the same data, because the additional memory overhead means that cache limits will be exceeded more often and memory paging will occur more frequently. For systems with processor designs that impose penalties for 16-bit aligned access but have very large memories, this effect may be less noticeable.~~

Characters Versus Code Points. In any event, Unicode code points do not necessarily match user expectations for "characters." For example, the following are not represented by a single code point: a combining character sequence such as <g, acute>; a conjoining jamo sequence for Korean; or the Devanagari conjunct "ksha." Because some Unicode text processing must be aware of and handle such sequences of characters as text elements, the fixed-width encoding form advantage of UTF-32 is somewhat offset by the inherently variable-width nature of processing text elements. See Unicode Technical Standard #18, "Unicode Regular Expressions," for an example where commonly implemented processes deal with inherently variable-width text elements owing to user expectations of the identity of a "character."

UTF-8. UTF-8 is reasonably compact in terms of the number of bytes used. It is really only at a significant size disadvantage when used for East Asian implementations such as Chinese, Japanese, and Korean, which use Han ideographs or Hangul syllables requiring three-byte code unit sequences in UTF-8. UTF-8 is also significantly less efficient in terms of processing than the other encoding forms.

Binary Sorting. A binary sort of UTF-8 strings gives the same ordering as a binary sort of Unicode code points. This is obviously the same order as for a binary sort of UTF-32 strings.

All three encoding forms give the same results for binary string comparisons or string sorting when dealing only with BMP characters (in the range U+0000..U+FFFF). However, when dealing with supplementary characters (in the range U+10000..U+10FFFF), UTF-16 binary order does not match Unicode code point order. This can lead to complications when trying to interoperate with binary sorted lists—for example, between UTF-16 systems and UTF-8 or UTF-32 systems. However, for data that is sorted according to the conventions of a specific language or locale rather than using binary order, data will be ordered the same, regardless of the encoding form.