Unicode—Logical and Visual Representations

David R. Mortensen

March 26, 2024

The Unicode Standard

The Unicode Standard is a way of encoding (almost) all known and historical human scripts in a consistent way. It is an ISO standard and is not the way almost all text on the internet and in major operating systems is ecoded.[1]

[1] Exceptions include certain language-specific versions of Microsoft Windows, which use legacy encodings for particular purposes.

The Bad Old Days

In 1963, the ASCII encoding, which would become the default encoding for computing in the English world,[2] was proposed. It was a 7-bit encoding with the first bit as a check bit. But what about languages other than English? Users and developers wanted to constrain wanted to constrain the number of bits per character but ASCII was not sufficient to represent languages written with diacritics and additional symbols over and above the English alphabet, even when they were written in a Latin script. As a result, a great number of ISO standards for eight-bit encodings were codified (see Table 1). Microsoft developed its own character sets, as did Apple. There were multiple competing standars for Korean, Japanese, and Chinese (e.g., Guobiao [=GB] for simplified characters on the Mainland and Big5, which was used for traditional characters in Taiwan). The ISCII character set was developed for languages of India.

[2] Despite the persistence of IBM's EBCDIC encoding.

Most of these are eight-bit encodings. However, encodings like GB had to represent single characters as multiple code points. Because each code point required eight bits, GB was prone to corruption (it was not autocorrecting, so if a byte was missing, the whole file or stream would be garbled).

This presented a nightmare for international computing. What if one wanted to create a text file with more than one language (more than one code page) represented? This was almost impossible.

The Structure of Unicode

Starting in the late 1980s, researchers at Xeros decided to do something about this untenable situation. They decided to create a universal character set. They following the following principles:

- Each character would be represented by a CODE POINT.
- The first seven bit-range would represent the ASCII character set.
- The first eight bit-range would represent Latin-1 (ISO 8859-1) (the first code page)

Table 1: Some eight-bit encodings defined in ISO 8859

ISO 8859-1	Western Europe
ISO 8859-2	Western and Central Europe
ISO 8859-3	Western Europe and South European (Turkish, Maltese plus Esperanto)
ISO 8859-4	Western Europe and Baltic countries (Lithuania, Estonia, Latvia and Lapp)
ISO 8859-5	Cyrillic alphabet
ISO 8859-6	Arabic
ISO 8859-7	Greek
ISO 8859-8	Hebrew
ISO 8859-9	Western Europe with amended Turkish character set
ISO 8859-10	Western Europe with rationalised character set for Nordic languages, including complete Icelandic set
ISO 8859-11	Thai
ISO 8859-13	Baltic languages plus Polish
ISO 8859-14	Celtic languages (Irish Gaelic, Scottish, Welsh)
ISO 8859-15	Added the Euro sign and other rationalisations to ISO 8859-1
ISO 8859-16	Central, Eastern and Southern European languages (Albanian, Bosnian, Croatian, Hungarian, Polish, Romanian, Serbian and Slovenian, but also French, German, Italian and Irish Gaelic)

- The first two byte (16 bit) range would represent the Basic Multilingal Plane (BMP), which included most commonly-used scripts in the world 1.
- There were 16 additional planes, which were represented using UTF-8 SURROGATES following by a sequence of bytes.

According to Wikipedia, "65,520 of the 65,536 code points in [the Basic Multilingual Plane] have been allocated to a Unicode block, leaving just 16 code points in a single unallocated range (2FE0..2FEF)"

Unicode Encodings

The Unicode Standard defines a mapping from GLYPHS (roughly, graphemes) and code points, but does not specify how these code points are to be encoded.

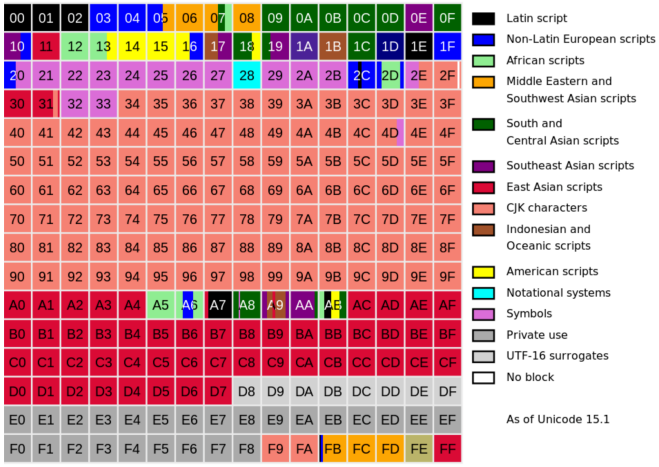

Figure 1: A map of the Basic Multilingual Plane. Each numbered box represents 256 code points (Wikipedia).

UCS-2

Universal Character Set 2 provided an early way of representing Unicode. All characters on the BMP were represented as two bytes. It could only represent characters in the BMP.

UTF-16 (16-bit Unicode Transformation Format)

UTF-16 was an extension of UTF-8. All characters in the BMP were represented as two bytes, but additional characters (in the other 16 planes) could be represented by surrogates, followed by additional bytes. Traditionally, it hsa been the internal representation of Unicode in Java and Python 2 and 3, although there is movement towards UTF-8.

UTF-32

UTF-32 simply uses four bytes per code point, allowing it to represent, roughly, 4,000,000,000 characters with a fixed number of bytes per code point. It is very space-inefficient but may be time-efficient.

UTF-1

This encoding, which is now not in common use, represented all characters in Unicode with sequences of byte of verious lengths (1–5 bytes). It was superseded by UTF-8.

UTF-8

If you only know one Unicode encoding, it should be UTF-8, the language of the Internet today. UTF-8 encodes characters as sequences of 1–4 bytes. All these sequences begin with the bits 0, 110, 1110,

or 1110. Susequent bytes in the stream start with 10 (see Figure 2. The prefixes mark the sequences as either the beginning of the code point or as a non-initial byte in the code point. This makes UTF-8 SELF-CORRECTING: even if some bytes are corrupted, it is always possible to recover and avoid corruption of the subsequent parts of the file.

The first seven-bit range in UTF-8 (U+0000 to U+007F) is identical to ASCII. This means that tools written for ASCII (like the original Unix command-line tools) can deal with UTF-8 without modification.[3]

[3] However, the Unix tools will treat each byte as a character, resulting in surprising results when counting characters

Code point ↔ UTF-8 conversion

First code point	Last code point	Byte 1	Byte 2	Byte 3	Byte 4
U+0000	U+007F	0xxxxxxx			
U+0080	U+07FF	110xxxxx	10xxxxxx		
U+0800	U+FFFF	1110xxxx	10xxxxxx	10xxxxxx	
U+010000	[b]U+10FFFF	11110xxx	10xxxxxx	10xxxxxx	10xxxxxx

Figure 2: Equivalences between Unicode code points and UTF-8

Now consider implications for bytewise byte-pair encoding. Take as an example, Lepcha (0+1C00–0+1CAF), show in Figure 3. The first two bytes, which are going to co-occur most frequently, encode the information about the columns, then the rows. Since Lepcha is unlikely to be well-represented in a corpus, This means that the encoding is going to group characters, in some sense, by the column (even when there is no logical relationship between characters in a column). This may or may not be desirable, depending on how code pages are shared among languages. In this case, the code page is shared with Chiki (in the columns) so the organization of Lepcha code points into columns is not undesirable, but the characters in the columns have little in common other than being Lepcha.

Rendering Unicode

In the old days, the visual representation of text followed directly from its logical representation—a sequence of bytes was represented by a monotonic sequence of corresponding glyphs (visual character shapes). This works fine for English (and other, very simple, scripts). However, as the range of scripts covered by Unicode has grown, more sophisticated technologies have been developed to represent text visually.

Bitmap versus Vector Fonts

The first FONTS[4] were collections of bitmap (raster) images.

[4] A font is a computer file containing a set of graphically related glyphs (Wikipedia).

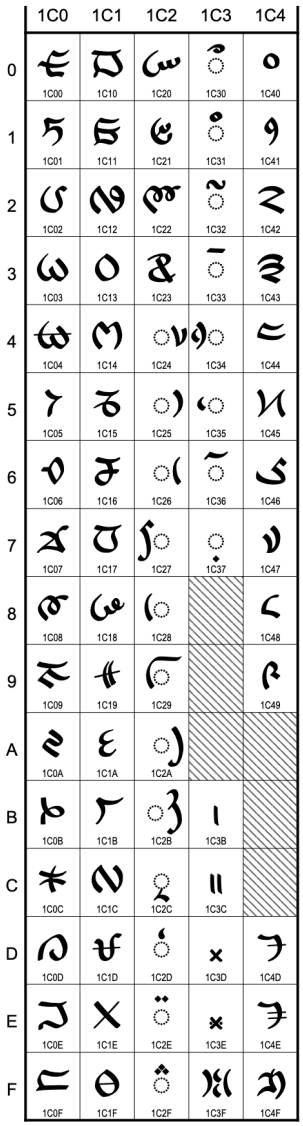

Figure 3: The Lepcha code page in the Unicode Standard

Because bitmap glyphs are easy to render, they are still frequently used for very low-level interactions with operating systems (for example, at the console when a graphical user interface is not yet, or cannot, be launched).

Vector fonts, or outline fonts, are represented as collections of vectors defining filled outlines. They can be scaled limitlessly without showing jagged edges. The vectors can also be manipulated and combined to make new characters from existing parts.

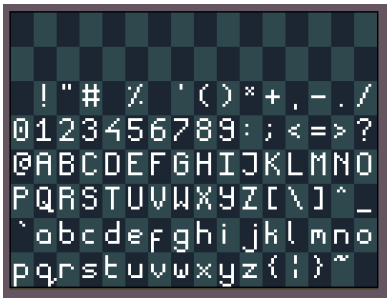

Figure 4: A bitmap font.

TrueType and OpenType

Some of the earliest vector fonts formats were Postscript font formats (Type 1 and Type 3). However, the formats that emerged victorious from the 1980s-1990s font wars were TrueType and OpenType. Both of these formats are still widely used. They define glyphs in terms of quadratic curves. They supported large character sets and included a kind of programming language for HINTING (rendering the glyphs according to context).

OpenType was a later format that was developed, in part, to deal with international fonts. They support much larger sets of glyphs and more sophisticated hinting.

These fonts can support COMBINING CHARACTERS—two or more characters that combine to form a single glyph.

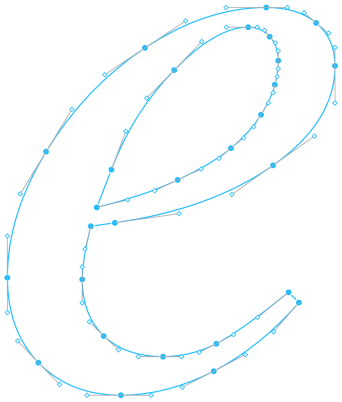

Figure 5: A vector glyph described in terms of bezier curves.

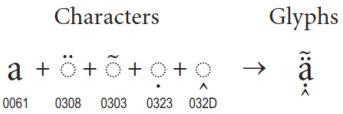

Figure 6: Exampe of five characters combining to form a single glyph.

Font-Level versus Application-Level Language Support

Fonts have to behave in different ways for different scripts and languages. There are two ways of achieving this: the logic can be provided at the level of the font or at the level of applications/operating systems. Implementing this at the level of the application reduces redundancy and makes fonts easier to develop. However, there is a problem: suppose a script is not (or not yet) supported by an application. This is often the case for minority orthographies. There is no way of retroactively adding this support to an application.

Two solutions to this problem are Apple Advanced Typography and SIL's Graphite.

Apple Advanced Typography (AAT)

AAT is a sophisticated typesetting technology that allows fonts to define a wide array of contextual, presentational, and script-based features. It is supported by extra TABLES added to TrueType and OpenType fonts. Only a limited number of fonts supporting AAT are available. Most of these are shipped with MacOS. AAT is mostly supported on MacOS but it is also supported on other platforms by XeTeX and LuaTeX.

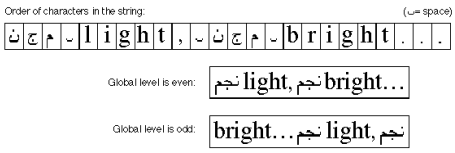

Figure 7: Example of Arabic rendering using AAT.

Graphite

Graphite is an open source alternative to AAT produced by SIL International. It is similar to AAT in some respects, but is supported by even fewer fonts (many of them produced by SIL)[5]. Like AAT, it can be incorporated into both TrueType and OpenType fonts. Support for Graphite is built in to many open source applications such as Mozilla Firefox, XeTeX, LuaTeX, OpenOffice, and Libre Office.

[5] See `https://graphite.sil.org/graphite_fonts.html`

References