

# Forms of Unicode

Mark Davis

In the beginning, Unicode was a simple, fixed-width 16-bit encoding. Under its initial design principles, there was enough room in 16 bits for all modern writing systems. But over the course of Unicode's growth and development, those principles had to give way. When characters were added to ensure compatibility with legacy character sets, available space dwindled rapidly. Many of these compatibility characters are superfluous, and were required only because different platform technologies at the time couldn't handle the representation of those characters as originally designed.

So 16 bits were not enough anymore. Unicode needed an extension mechanism to get up to a larger number of characters. The standard mechanism uses pairs of Unicode values called

*surrogates* to address over 1,000,000 possible values. There are no characters in surrogate space yet, although there should be by the end of the year 2000.

Additionally, some systems couldn't easily handle extending their interfaces to use 16-bit units in processing. These systems needed a form of Unicode that could be handled in 8-bit bytes. Other systems found it easier to use larger units of 32 bits for representing Unicode.

As a result of these different requirements, there are now three different forms of Unicode: UTF-8, UTF-16, and UTF-32. (See Acronyms defined later in this article.) A great deal of thought was put into the development of these three forms, so that each form is most useful for its particular environment, and each can be quickly converted to and from the other forms as needed. It is important to understand both the capabilities -- and the limitations -- of each of these forms.

UTF-32 is a working name from *UTR #19: Interoperable 32-bit Serialization*. It is very similar to the ISO 10646 format *UCS-4*, except that it is constrained to valid Unicode values for interoperability. The Unicode Standard similarly constrains the Unicode definition of UTF-8.

## Characters vs. Glyphs

Before getting into these forms of Unicode, we need to make a few things clear. First is the distinction between *characters* and *glyphs*. A glyph is a particular image that represents a character or part of a character. It may have very different shapes: below are just some of the possibilities for the letter *a*. In the examples below, a selection of alternatives is presented in different cells in the table.

Character	Sample glyphs					
a	ɑ	Ɑ	ⱪ	ⱥ	ⱦ	Ⱨ

Glyphs do not correspond one-for-one with characters. For example, a sequence of *f* followed by *i* can be represented with a single glyph, called an *fi ligature*. Notice that the shapes are merged together, and the dot is missing from the *i*.

Character sequence	Sample glyph
f i	fi

## Forms of Unicode

Mark Davis

On the other hand, the same image as the *fi* ligature could be achieved by a sequence of two glyphs with the right shapes. The choice of whether to use a single glyph or a sequence of two is up to the font containing the glyphs and the rendering software.

Character sequence	Possible glyph sequence
f i	f l

Similarly, an accented character could be represented by a single glyph, or by different component glyphs positioned appropriately. In addition, the separate accents can also be considered characters in their own right, in which case a sequence of characters can also correspond to different possible glyph representations:

Character Sequence	Possible glyph sequences			
ô	ô	o ^ `	o ^	
o ^ `	ô	o ^ `	o ^	

In non-Latin languages, the connection between glyphs and characters may be even less direct. Glyphs may be required to change their shape and widths depending on the surrounding glyphs. These glyphs are called contextual forms. For example, see the Arabic glyphs below.

Character	Possible Glyphs, depending on context				
o	o	f	o	a	

Glyphs may also need to be widened for justification instead of simply adding width to the spaces. Ideally this would involve changing the shape of the glyph depending on the desired width. On some systems, this widening can be achieved by inserting extra connecting glyphs called *kashidas*. In such a case, a single character can conceivably correspond to a whole sequence of *kashidas* + *glyphs* + *kashidas*.




Character	Sequence of glyphs
o	— f —

In other cases a single character must correspond to two glyphs, because those two glyphs are positioned *around* other letters. See the Tamil characters below. If one of those glyphs forms a

## Forms of Unicode

Mark Davis

ligature with other characters, then we have a situation where *part* of a character corresponds to *part* of a glyph. If a character (or any part of it) corresponds to a glyph (or any part of it), then we say that the character *contributes* to the glyph.

Character	Split glyphs
	 

The upshot is that the correspondence between glyphs and characters is not one-to-one, and cannot in general be predicted from the text. The ordering of glyphs will also not in general correspond to the ordering of the characters, because of right-to-left scripts like Arabic and Hebrew. Whether a particular string of characters is rendered by a particular sequence of glyphs will depend on the sophistication of the host operating system and the font.













### Characters vs. Code Points vs. Code Units

In principle, the goal of Unicode is pretty simple: to assign each character in the world a number, called a *code point* (or *scalar value*). These numbers can range from 0 to 1,114,111 = 10FFFF<sub>16</sub>, although every value of the form xxFFFE<sub>16</sub> or xxFFFF<sub>16</sub> is illegal.

Given this goal, we can just say that Unicode encodes characters, right?

*Wrong.* One of the main design requirements on Unicode was to interwork with pre-existing character encoding standards. This requires lossless round-tripping, where you can take text from an encoding into Unicode, and back again, without losing information. Unfortunately, those legacy character encoding standards contained a lot of things that aren't characters, and Unicode had to pull them in.

These non-characters included ligature glyphs, contextual form glyphs, glyphs that varied in width, sequences of characters, and adorned glyphs (such as circled numbers). The following examples show where glyphs are encoded as single characters in Unicode. As with glyphs, there is no one-to-one relationship between characters and code points. What an end-user thinks of as a single character (*grapheme*) may in fact be represented by multiple code points; conversely, a single code point may correspond to multiple characters. Here are some examples.

Characters	Unicode Code Points	Notes
	   	<i>Each of these Arabic contextual form glyphs has a code point.</i>
 		<i>This ligature glyph has a code point.</i>
  		<i>A single code point represents a sequence of three characters.</i>

# Forms of Unicode

Mark Davis



Even once we have code points, we're not done. A particular encoding will represent code points as a sequence of one or more *code units*, where a code unit is a unit of memory: 8, 16, or 32 bits. We will talk more about this when we get to "Memory formats."

## Avoiding Ambiguity

We have seen that *characters*, *glyphs*, *code points*, and *code units* are all different. Unfortunately the term *character* is vastly overloaded. At various times people can use it to mean any of these things:

- An image on paper (glyph)
- What an end-user thinks of as a character (grapheme)
- What a character encoding standard encodes (code point)
- A memory storage unit in a character encoding (code unit)

Because of this, ironically, it is best to avoid the use of the term *character* entirely when discussing character encodings, and stick to the term *code point*.

## Memory Formats

Now that we have gotten all that out of the way, let's turn back to the Unicode formats. For now, we will discuss only the use of UTFs in memory (there is an additional complication when it comes to serialization, but we will get to that later). Each UTF-*n* represents a code point as a sequence of one or more *code units*, where each code unit occupies *n* bits.

Table 1 shows the code unit formats that the UTFs use, and provides an indication of the storage requirements averaged over all computer text. The average storage requirements for UTF-16 and UTF-32 should remain constant over time (I would be surprised if the average number of UTF-16 surrogate-space characters ever hit 0.1%).

However, the average storage requirements for UTF-8 will change, because it is heavily dependent on the proportion of text in different languages. The growth between the years 1999 and 2004 is due to the expected increase in the use of computers in East and South Asia. (The estimates in the table use the proportion of Web page languages in the world as a proxy for all computer text. Those proportions are based on estimates from IDC.)

# Forms of Unicode

Mark Davis

Table 1: UTF Types

UTF	Formats	Estimated average storage required per page (3000 characters)	
UTF-8		3 KB (1999)	On average, English takes slightly over one unit per code point. Most Latin-script languages take about 1.1 bytes. Greek, Russian, Arabic and Hebrew take about 1.7 bytes, and most others (including Japanese, Chinese, Korean and Hindi) take about 3 bytes. Characters in surrogate space take 4 bytes, but as a proportion of all world text they will always be very rare.
		5 KB (2003)	
UTF-16		6 KB	All of the most common characters in use for all modern writing systems are already represented with 2 bytes. Characters in surrogate space take 4 bytes, but as a proportion of all world text they will always be very rare.
UTF-32		12 KB	All take 4 bytes

Programming using UTF-8 and UTF-16 is much more straightforward than with other mixed-width character encodings. For each code point, they have either a singleton form or a multi-unit form. With UTF-16, there is only one multi-unit form, having exactly two code units. With UTF-8, the number of trailing units is determined by the value of the lead unit: thus you can't have the same lead unit with a different number of trailing units. Within each encoding form, the values for singletons, for lead units, and for trailing units are all *completely disjoint*. This has crucial implications for implementations:

- No overlap. If you search for string A in a string B, you will *never* get a false match on code points. You *never* need to convert to code points for string searching. False matches never occur because the end of one sequence can never be the same as the start of another sequence. *Overlap is one of the biggest problems with common multi-byte encodings like Shift-JIS. All of the UTFs avoid this problem.*
- Determinate boundaries. If you randomly access into text, you can *always* determine the nearest code-point boundaries with a small number of machine instructions.
- Pass-through. Processes that don't look at particular character values don't need to know about the internal structure of the text.
- Simple iteration. Getting the next or previous code point is straightforward, and only takes a small number of machine instructions.
- Slow indexing. Except in UTF-32, it is inefficient to find code unit boundaries corresponding to the *n*th code point, or to find the code point offset containing the *n*th code unit. Both involve scanning from the start of the text.
- Frequency. Because the proportion of world text that needs surrogate space is extremely small, UTF-16 code should always be optimized for the single code unit. With UTF-8, it is probably worth optimizing for the single-unit case also, but not if it slows down the multi-unit case appreciably.

UTF-8 has one additional complication, called the *shortest form requirement*. Of the possible sequences in Table 1 that could represent a code point, the Unicode Standard requires that the

# Forms of Unicode

Mark Davis

shortest possible sequence be generated. When mapping back from code units to code points, however, implementations are not required to check for the shortest form. This problem does not occur in UTF-16.

Most systems will be upgrading their UTF-16 support for surrogates in the next year or two. This upgrade can use a phased approach. From a market standpoint, the only interesting surrogate-space characters expected in the near term are an additional set of CJK ideographs used for Japan, China, and Korea. If a system is already internationalized, most of the operations on the system will work sufficiently well that minor changes will suffice for these in the near term.

## Storage vs. Performance

Both UTF-8 and UTF-16 are substantially more compact than UTF-32, when averaging over the world's text in computers. UTF-8 is currently more compact than UTF-16 on average, although it is not particularly suited for East-Asian text because it occupies about 3 bytes of storage per code point. UTF-8 will probably end up as about the same as UTF-16 over time, and may end up being less compact on average as computers continue to make inroads into East and South Asia. Both UTF-8 and UTF-16 offer substantial advantages over UTF-32 in terms of storage requirements.

Code-point boundaries, iteration, and indexing are very fast with UTF-32. Code-point boundaries, accessing code points at a given offset, and iteration involve a few extra machine instructions for UTF-16; UTF-8 is a bit more cumbersome. Indexing is slow for both of them, but in practice indexing by different code units is done very rarely, except when communicating with specifications that use UTF-32 code units, such as XSL.

This point about indexing is true unless an API for strings allows access only by *code point* offsets. This is a very inefficient design: strings should always allow indexing with *code unit* offsets. Moreover, because code points do not, in general, correspond to end-user expectations for characters, it is often better to use *grapheme* (user character) boundaries instead, and to store text in strings rather than as single code points. See *The Unicode Standard, Version 3.0* for more information.

Conversion between different UTFs is very fast. Unlike converting to and from legacy encodings like Latin-2, conversion between UTFs doesn't require table lookups.

Overall performance may also be affected by other factors. For example, if the code units match the machine word size, access can be faster; but if they use up more memory, more page faults and cache faults may occur, slowing down performance.

## Serialized Formats

Serialization is the process of converting a sequence of code units into a sequence of bytes for storage or transmission. There are two complications with serialization, *endianness* and *encoding signatures*:

- Endianness. If a code unit is not a single byte, it can be written in two ways because of differences in machine architectures: big endian (most significant byte first) or little endian (least significant byte first). With today's microprocessor speed this is not a big deal, but at the time Unicode was being adopted it was felt that both BE and LE formats were required.
- Encoding. If a system does not tag files with the character encoding, then it might know that the file contains text, but not know which encoding is used.

# Forms of Unicode

Mark Davis

To meet these two requirements (from an unnamed, but rather influential company), the character ZERO WIDTH NOBREAK SPACE (FEFF 16) can be used as a *signature* in the initial few bytes of a file. When the character has that usage, it is called a *byte order mark (BOM)*. The BOM has the special feature that its byte-swapped counterpart BSBOM (FFFE) is defined to never be a valid Unicode character, so it also serves to indicate the endianness. This signature is not part of the content -- think of it as a mini-header -- and must be stripped when processing. For example, blindly concatenating two files will give an incorrect result.

Because of this, there are multiple forms of the UTFs, based on their endianness and whether they use a signature (see Table 2 for details). It is important not to confuse the memory formats and serialized formats, especially because some of them have the same names. For example, *UTF-16* when referring to a memory format has no endianness issues and does not use a signature (one may be present when generating or reading files, but this should be only ephemeral). On the other hand, *UTF-16* when referring to a serialization format may have a signature and may be in either endianness.

Now, most people will never need to know all the grimy details of these encodings, but Table 2 shows the differences for those who are interested. In practice, most of these are handled transparently by the character-code conversion utilities.

**Table 2: UTF Serializations**

<b>UTF-8</b>	<ul style="list-style-type: none"><li>▪ Initial EF BB BF is a signature, indicating that the rest of the file is UTF-8.</li><li>▪ Any EF BF BE is an error.</li><li>▪ A real ZWNBSP at the start of a file requires a signature first.</li></ul>
<b>UTF-8N</b>	<ul style="list-style-type: none"><li>▪ All of the text is normal UTF-8; there is no signature.</li><li>▪ Initial EF BB BF is a ZWNBSP.</li><li>▪ Any EF BF BE is an error.</li></ul>
<b>UTF-16</b>	<ul style="list-style-type: none"><li>▪ Initial FE FF is a signature indicating the rest of the text is big endian UTF-16.</li><li>▪ Initial FF FE is a signature indicating the rest of the text is little endian UTF-16.</li><li>▪ If neither of these are present, all of the text is big endian.</li><li>▪ A real ZWNBSP at the start of a file requires a signature first.</li></ul>
<b>UTF-16BE</b>	<ul style="list-style-type: none"><li>▪ All of the text is big endian: there is no signature.</li><li>▪ Initial FE FF is a ZWNBSP.</li><li>▪ Any FF FE is an error.</li></ul>
<b>UTF-16LE</b>	<ul style="list-style-type: none"><li>▪ All of the text is little endian: there is no signature.</li><li>▪ Initial FF FE is a ZWNBSP.</li><li>▪ Any FE FF is an error.</li></ul>
<b>UTF-32</b>	<ul style="list-style-type: none"><li>▪ Initial 00 00 FE FF is a signature indicating the rest of the text is big endian UTF-32.</li><li>▪ Initial FF FE 00 00 is a signature indicating the rest of the text is little endian UTF-32.</li><li>▪ If neither of these are present, all of the text is big endian.</li></ul>

# Forms of Unicode

Mark Davis

	<ul style="list-style-type: none"><li>▪ A real ZWNBSP at the start of a file requires a signature first.</li></ul>
<b><i>UTF-32BE</i></b>	<ul style="list-style-type: none"><li>▪ All of the text is big endian: there is no signature.</li><li>▪ Initial 00 00 FE FF is a ZWNBSP.</li><li>▪ Any FF FE 00 00 is an error.</li></ul>
<b><i>UTF-32LE</i></b>	<ul style="list-style-type: none"><li>▪ All of the text is little endian: there is no signature.</li><li>▪ Initial FF FE 00 00 is a ZWNBSP.</li><li>▪ Initial 00 00 FE FF is an error.</li></ul>

**Note:** *The italicized names are not yet registered, but are useful for reference.*

## Decisions, Decisions...

Ultimately, the choice of which encoding format to use will depend heavily on the programming environment. For systems that only offer 8-bit strings currently, but are multi-byte enabled, UTF-8 may be the best choice. For systems that do not care about storage requirements, UTF-32 may be best. For systems such as Windows, Java, or ICU that use UTF-16 strings already, UTF-16 is the obvious choice. Even if they have not yet upgraded to fully support surrogates, they will be before long.

If the programming environment is not an issue, UTF-16 is recommended as a good compromise between elegance, performance, and storage.

## Acronyms Defined

For those unfamiliar with some of the acronyms used in this article, here is a short list.

<b>ICU</b>	<b>IBM Components for Unicode: IBM open-source software for Unicode enablement</b>
<b>IDC</b>	<b>International Data Corporation: a provider of information technology data</b>
<b>UTF</b>	<b>Unicode Transformation Format: a transformation that maps each Unicode code point onto a unique sequence of code units.</b>  <b>Note: UTF-32 is a working name from <i>UTR #19: Interoperable 32-bit Serialization</i>. It is very similar to the ISO 10646 format <i>UCS-4</i>, except that it is constrained to valid Unicode values for interoperability. The Unicode Standard similarly constrains the Unicode definition of UTF-8.</b>
<b>W3C</b>	<b>World Wide Web Consortium: organization responsible for HTML, XML and other Web standards.</b>
<b>XML</b>	<b>Extensible Markup Language: the "universal format for structured documents and data on the Web"</b>
<b>XSL</b>	<b>Extensible Stylesheet Language: one of the many specifications in the XML family being developed by the W3C</b>