A brief introduction to code pages and Unicode

How Unicode relates to code pages, character sets, and encoding

Markus Scherer
March 03, 2000

Unicode is intended to unify the computing community around a single standard for encoding text. To understand how the standard works and why we need it, explore the code behind the letters you see on your screen and in your printouts.

What is text?

Concepts like code page and encoding describe the way text is stored in computers, in files and data structures, and how applications handle such text. When you use a computer to write and file your master's thesis or your mother's Black Forest cake recipe, you produce text that you expect your computer to store, to display on your home page, or to send in e-mail. You want to be able to search for a word, copy and paste pieces of sentences, and so on.

Inside a computer program or data file, text is stored as a sequence of numbers, just like everything else. These sequences are integers of various sizes, values, and interpretations, and it is the code pages, character sets, and encodings that determine how integer values are interpreted.

Text consists of characters, mostly. Fancy text or rich text includes display properties like color, italics, and superscript styles, but it is still based on characters forming plain text. Sometimes the distinction between fancy text and plain text is complex, and the distinction may depend on the application. Here, we focus on plain text.

So, what is a character? Typically, a letter. Also, a digit, a period, a hyphen, punctuation, and math symbols. There are also control characters (typically not visible) that define the end of a line or paragraph. There is a character for tabulation, and a few others in common use.

Now that we know what a character is, what number is assigned to each one? This is where it gets interesting: It depends!

A simple character such as the letter "a" may have different integer values in different programs or data files. In some instances, there may not even be a number for a certain character. The integers used for characters have different sizes, or numbers of bits. If the character is really an "ä", an "a"
with dots above it, then it might be stored as two characters with two integer values; one for the "a" and one for the dots.

**Character sets: characters with numbers**

Integers used in computing always come with upper limits, depending on the number of bits that are used to store such an integer. This determines how many different characters you can distinguish at a time.

Imagine that you are designing a character set. First, you decide how many and which characters you need, and set an upper limit for the integer values. These characters are the repertoire that you will work with. Then you give each character a number (typically a unique one, but occasionally several), and void, you've got a character set. The result is called a Coded Character Set; before you assign the numbers, the collection of characters is called an Abstract Character Repertoire.

How big a repertoire do you need? For the English alphabet, with some digits and little more, maybe around 60 characters. The Western European Teletex standard comes with about 330 characters for the many languages. Korean has almost 12,000 syllables, and some comprehensive Chinese dictionaries list far more than 50,000 letters in their script. There are also hundreds of other characters in common use, such as math and currency symbols.

Historically, computers were pretty slow and had fairly little memory, and they were very expensive. To make matters worse, up into the 1960s, getting any text or data into a computer meant punching holes into pieces of paper and feeding stacks of them into the computer. Some of our character sets date back to that punch-card age and are designed with these cards in mind. In fact, most of the character sets that we have to this day are based on those 1960s design decisions!

In the early days of computers, every computer maker invented their own machine and memory layout, and there were different popular machine word sizes. At first, this wasn't a problem, because there was no Internet where everything needed to fit together -- every vendor just did what fit their customers. As a result, there was a great variety of bits per byte and bits per machine word, and different computer architectures came with different character sets and encodings. Characters were stored with anywhere from 5 to 9 bits each.

The two character set dinosaurs that are still roaming the circuits of the networks are ASCII and EBCDIC, both from the 1960s. Where there is still a Telex (TTY) terminal, there is also the much older Baudot-code. Baudot was designed for 5-bit units, ASCII for 7 bits, and EBCDIC for 8 bits. Another important legacy from those days is the fact that some of the Internet e-mail system is still only prepared to handle 7-bit bytes. Fortunately, 7-bit e-mail gateways are a dying species. Every modern computer architecture uses bytes and machine words with at least 8 bits and that are powers of 2 (8, 16, 32, 64, and so on).
Character sets, bytes, and encodings

When the character set units fit into single bytes, the encoding is trivial and indistinguishable from the character set itself. For character sets with units that are larger than bytes, there are often several encodings to fit different needs, and one single encoding might carry characters from more than one character set to make them even more versatile. Trying to be compatible with 7-bit byte machines limited many encodings to just use 7 bits, which wastes an 8th of the memory available in modern, 8-bit byte-based machines.

ASCII: The American Standard Code for Information Interchange

From the point of view of the Internet Age, ASCII and other codes from the early period of computing had too many control codes, and some with interesting semantics. ASCII provides only 128 numeric values, and 33 of those are reserved for special functions.

For example, there are two "remove" characters, Backspace and Delete. The Backspace is from Teletype writers, where it moves the typing head back and allows a following letter to overlap the previous one for underlining and accent marks. Later, it was used as the code that a keyboard sends when the Backspace key is pressed. Modern text processing does not need to use a character value for this -- it is better to use a more precise protocol.

The Delete is from punch-card days. It has its unique value, 0x7f, because when someone punched a wrong code on a card, he would punch out all the holes in that column to delete the wrong letter. "All holes in a column" was read as all one-bits, and an integer with seven one-bits has the value 0x7f, or 127!

Another unhelpful legacy is having Teletype-controls for Carriage Return ("move the type head to the beginning of the current line") and Line Feed ("move the paper forward by one line") instead of semantic controls for "end of paragraph" and "end of line". This still causes confusion in the exchange of text today.

Many other controls were designed for Teletype functions, as protocol bytes in serial communications with primitive modems, field separators in databases, and other nontext functions.

Only 95 ASCII code points are used for "real" text-characters (or 94, not counting the space character). These graphic characters are mostly Latin upper- and lower-case letters, digits, and punctuation, plus some special braces, an underline, and some accent marks. It is a good base for the American market, but not for European languages with their accented letters, and does not cover any other scripts.

EBCDIC: The Extended Binary-Coded Decimal Interchange Code

IBM designed this encoding format and a number of character sets using this format for its mainframes, using 8-bit bytes. It was developed at a similar time as ASCII, with some similar properties. It too had many (65 out of 256) control code positions. Unlike ASCII, the Latin letters are not combined in two blocks for upper- and lower-case. Instead, the letters are arranged so that their hexadecimal values have second digits of 1 through 9 -- another punch-card-friendly design.
Character sets after ASCII and before Unicode

It is important to realize the impact of ASCII on the design of character sets and encodings in the 1970s and 1980s. Many of them were designed to be modifications or extensions of ASCII, and especially to use only the 94 graphic character codes that ASCII provides.

Encodings that were designed to stay within 7 bits per byte started using up to two such bytes with each, only using the 94 graphic positions. Other encodings used 8-bit bytes, but often only the 94 positions that correspond to the ASCII graphic codes (adding 0x80=128), sometimes leaving the other 34 code points for another Delete, 32 even more rare control codes, and a nonbreaking space. Over time, 8-bit bytes were used with up to two, and later, up to four, bytes per code point, with many reserved or control codes reducing the useful encoding space.

Character sets for many characters

The most common encodings (character encoding schemes) use a single byte per character, and they are often called single-byte character sets (SBCS). They are all limited to 256 characters. Because of this, none of them can even cover all of the accented letters for the Western European languages. Consequently, many different such encodings were created over time to fulfill the needs of different user communities.

The most widely used SBCS encoding today, after ASCII, is ISO-8859-1. It is an 8-bit superset of ASCII and provides most of the characters necessary for Western Europe. A modernized version, ISO-8859-15, also has the euro symbol and some more French and Finnish letters.

Double-byte character sets (DBCS) were developed to provide enough space for the thousands of ideographic characters in East Asian writing systems. Here, the encoding is still byte-based, but each two bytes together represent a single character.

Even in East Asia, text contains letters from small alphabets like Latin or Katakana. These are represented more efficiently with single bytes. Multi-byte character sets (MBCS) provide for this by using a variable number of bytes per character, which distinguishes them from the DBCS encodings. MBCSs are often compatible with ASCII; that is, the Latin letters are represented in such encodings with the same bytes that ASCII uses. Some less often used characters may be encoded using three or even four bytes.

An important feature of MBCSs is that they have byte value ranges that are dedicated for lead bytes and trail bytes. Special ranges for lead bytes, the first bytes in multibyte sequences, make it possible to decide how many bytes belong together to encode a single character. Traditional MBCS encodings are designed so that it is easy to go forwards through a stream of bytes and read characters. However, it is often complicated and very dependent on the properties of the encoding to go backwards in text: going backwards, it is often hard to find out which variable number of bytes represents a single character, and sometimes it is necessary to go forward from the beginning of the text to do this.

Examples of commonly used MBCS encodings are Shift-JIS and EUC-JP (for Japanese), with up to two and three bytes per character, respectively.
Stateful encodings

Some encodings are stateful; they have bytes or byte sequences that switch the meanings of the following bytes. Simple encodings use Shift-In and Shift-Out control characters (bytes) to switch between two states. Sometimes, the bytes after a Shift-In are interpreted as a certain SBCS encoding, and the bytes after a Shift-Out as a certain DBCS encoding. This is very different from an MBCS encoding where the bytes for each character indicate the length of the byte sequence.

The most common stateful encoding is ISO 2022 and its language-specific variations. It uses Escape sequences (byte sequences starting with an ASCII Escape character, byte value 27) to switch between many different embedded encodings. It can also "announce" encodings that are to be used with special shifting characters in the embedded byte stream. Language-specific variants like ISO-2022-JP limit the set of embeddable encodings and specify only a small set of acceptable Escape sequences for them.

Such encodings are very powerful for data exchange but hard to use in an application. Their flexibility allows you to embed many other encodings, but direct use in programs and conversions to and from other encodings are complicated. For direct use, a program has to keep track not only of the current position in the text, but also of the state -- which embeddable encoding is currently active -- or must be able to determine the state for a position from considerable context. For conversions to other encodings, converting software may need to have mappings for many embeddable encodings, and for conversions from other encodings, special code must figure out which embeddable encoding to choose for each character.

Unicode: The last character set?

The Unicode standard specifies a character set and several encodings. As of early 2000, it contains almost 50000 characters, which include all the characters of the common character sets that were in use when Unicode was started around 1990, plus many that have been added since. It is an open character set, which means that it keeps growing and adding less frequently used characters.

The standard assigns numbers from 0 to 0x10FFFF, which is more than a million possible numbers for characters. About 5% of this space is used. Another 5% is in preparation, about 13% is reserved for private use (anyone can place any character in there), and about 2% is reserved and not to be used for characters. The remaining 75% is open for future use but not by any means expected to be filled up. In other words, there is finally a character set with plenty of space!

Unicode is in use today, and it is the preferred character set for the Internet, especially for HTML and XML. It is slowly being adopted for use in e-mail, too. Its most attractive property is that it covers all the characters of the world (with exceptions, which will be added in the future). Unicode makes it possible to access and manipulate characters by unique numbers -- their Unicode code points -- and use older encodings only for input and output, if at all.

Why Unicode?

Hundreds of encodings have been developed, each for small groups of languages and special purposes. As a result, the interpretation of text, input, sorting, display, and storage depends on the
knowledge of all the different types of character sets and their encodings. Programs are written to either handle one single encoding at a time and switch between them, or to convert between external and internal encodings.

Part of the problem is that there is no single, authoritative source of precise definitions of many of the encodings and their names. Transferring of text from one machine to another one often causes some loss of information. Also, if a program has the code and the data to perform conversion between a significant subset of traditional encodings, then it carries several Megabytes of data around.

Unicode provides a single character set that covers the languages of the world, and a small number of machine-friendly encoding forms and schemes to fit the needs of existing applications and protocols. It is designed for best interoperability with both ASCII and ISO-8859-1, the most widely used character sets, to make it easier for Unicode to be used in applications and protocols.

**Unicode encodings**

For single characters, 32-bit integer variables are most appropriate for the value range of Unicode. For strings, however, storing 32 bits for each character takes up too much space, especially considering that the highest value, 0x10FFFF, takes up only 21 bits. 11 bits are always unused in a 32-bit word storing a Unicode code point. Therefore, you will find that software generally uses 16-bit or 8-bit units as a compromise, with a variable number of code units per Unicode code point. It is a trade-off between ease of programming and storage space.

As a result, there are three common ways to store Unicode strings:

- UTF-32, with 32-bit code units, each storing a single code point
- UTF-16, with one or two 16-bit code units for each code point
- UTF-8, with one to four 8-bit code units (bytes) for each code point

UTF-16 is extremely well designed as the best compromise between handling and space, and all commonly used characters can be stored with one code unit per code point, where the code unit actually has the same integer value as the code point. This is the default encoding for Unicode.

UTF-8 is used mainly as a direct replacement for older MBCS encodings which all use 8-bit code units, but it takes some more code to process it.

For files and networks, where a string is written as a stream of bytes, it is important to know for 16- and 32-bit code units whether the most or least significant byte is written first. Thus, for byte streams, both UTF-16 and UTF-32 need to be specified as big-endian (most significant byte first) or little-endian (least significant byte first). Big-endian is the preferred network byte order as defined in Internet protocols. So, for example, you will see two versions of UTF-32; UTF-32BE and UTF-32LE.

If it is more important to save space than to efficiently access a random position in a byte stream, then the Standard Compression Scheme for Unicode (SCSU) should be considered: It is byte-
based and allows storage of text with about as few bytes as legacy encodings. It combines the universal character set with storage efficiency, and it is fairly easily converted to and from the UTFs. It is, however, a stateful encoding, and not suited for internal processing.

There is more exact terminology: internal encodings, with the byte ordering determined by the machine architecture, are called Character Encoding Forms. External encodings for byte streams are called Character Encoding Schemes.

Software that was designed for versions of Unicode before 2.0 (published in 1996) may be designed only for 16-bit code points, with a fixed-length UCS-2 instead of the variable-length UTF-16, and expecting at most three bytes per code point when using UTF-8. Since Unicode 2.0, when it was clear that 16 bits per code point were not enough, UTF-16 is the default encoding.
Related topics

- For another overview, see the International Unicode Conference 13, Tutorial TA4, "Survey of Character Encodings" by Peter K. Edberg pedberg@apple.com.
- Visit the Unicode site for more on character sets and encodings.
- The Unicode Consortium provides information about the Unicode Standard, and about how to order the book describing the standard.
- Find out more about SCSU, a Standard Compression Scheme for Unicode.
- Interested in Linux? See Unicode for Unix/Linux.
- Download IBM product evaluation versions and get your hands on application development tools and middleware products from DB2®, Lotus®, Rational®, Tivoli®, and WebSphere®.