To Unicode or Not to Unicode

The decision rests on the answers to some basic questions about your international software

KEN LUNDE

With the rapid integration of Unicode into various software products, such as applications and entire operating systems, one can only think to ask specific questions. Is Unicode really the right choice for my software development needs? What specific problems does Unicode solve? What new problems does Unicode create? Does Unicode spell an end to legacy encodings? These are all fair and reasonable questions to ask, and all of them have answers.

It is by no means my intention to shed any bad light on Unicode in the context of this article, but rather to make developers better aware of all the important issues, specifically as they relate to CJKV (Chinese, Japanese, Korean and Vietnamese) software development. Ignorance is a dangerous thing in the software industry, especially when it comes to internationalization and localization. Making truly wise decisions means that you must be armed with something far less dangerous, specifically knowledge and understanding.

Is Han Unification Problematic?

The answer depends on what type of software you plan to develop. Han Unification is described as the process of combining the Chinese characters (called hanzi, kanji or hanja, depending on the language) from numerous national standards, which resulted in a set of 20,902 characters (plus an additional 302 for compatibility purposes). Characters that are considered "common" across CJKV locales were given a single (unified) code point. Otherwise, well over 100,000 code points would have been required!

If you only need to deal with a single locale at a time, or if you are building font products for single locales, then Han Unification does not present a problem. Not at all. This is important to understand. All of the characters that you need should be available in Unicode, and the typeface design specifies the glyphs that are used. You will deal with the same characters as before, but they will simply be encoded differently. That is, encoded according to Unicode. Windows TrueType fonts have been Unicode-encoded (using a Unicode 'cmap' table) since Windows 95. There are well-established mapping tables that enable conversion between Unicode and legacy encodings.

If you need to deal with multiple locales (including single font products that support them), there is a problem. However, there is a solution to every problem, and this is by no means an exception. I suggest that you read Dirk Meyer's article, also in this issue, for specific methods for addressing the problem of creating Unicode CJKV fonts suitable for multiple locales.

Must Legacy Encodings Be Supported?

Yes. It is an absolute requirement. It is simply unavoidable. Legacy encodings will continue to exist far into the future (after all, that is why they are referred to as "legacy" encodings). It is important to support conversion to and from legacy encodings. Not all text will be in Unicode, and to process such text with a Unicode application, it must be converted to Unicode. Likewise, if the text you are processing needs to be used on non-Unicode systems, it must be converted into a legacy encoding.

Programming languages such as Java include a very rich set of built-in code converters that handle the conversion to and from legacy encodings — with the greatest of ease. I might add. Basis Technology's Rosette is a C++ library for handling Unicode, including code conversion with legacy encodings.

Does Unicode Include Every Character?

No. It cannot. No character set can possibly include every character ever created. New characters are still being created today, but it is a slow and hardly noticeable process. Even considering Ideographic Extension A (6,582 additional Chinese characters) that will become part of Unicode Version 3.0, there are still plenty of character set standards that do not have complete coverage in Unicode. Some examples to consider: CNS 11643-1992 has nearly 50,000 hanzi, fewer than half of them in Unicode. Hong Kong's GCCS includes over 3,000 hanzi, and approximately half of them are not in Unicode.

What Problems Does Unicode Solve?

Unicode effectively puts all characters on the same playing field or level, specifically that all characters are given equal treatment. Latin character *A* (U+0041) is represented with the same number of bits (or bytes, if the distinction matters to you) as the most complex Chinese character in Unicode (U+9F98). All characters are represented by 16 bits (or two 16-bit entities in the case of UTF-16 surrogates).

Characters that could not be used together in a single font in the past can now live in harmony. Japanese fonts, most of which are based on Shift-JIS encoding, include a highly restricted set of Latin characters. Basically, such fonts include only ASCII in the single-byte range. No special types of punctuation (smart quotes, the various dashes, ligatures and so on), and no accented Latin characters. Unicode overcomes this limitation, allowing a single font to include any type of character.

Unicode is a superset of the most common national character set standards, and can be effectively used as an interchange code

between them. I took advantage of this characteristic by developing a Unicode-based code converter called CJKVConv.pl (written in Perl). Converting between multiple encodings requires many mapping tables. For example, if you needed to convert between 6 encodings, you would need n^2 (36) mapping tables; but if Unicode were used as the interchange code, only 2n (12) mapping tables would be required. A significant savings of resources.

Unicode is an evolving character set, which typically means that more and more characters are being added to its now rather large repertoire. Currently, only Plane 0 (zero) is occupied, with over 40,000 characters (there will be slightly more when Unicode Version 3.0 is released). It is also known as the Basic Multilingual Plane (BMP). There are plans to use Plane 1 for encoding additional alphabetic characters, and Plane 2 for encoding additional Chinese characters.

What New Problems Does Unicode Create?

The "problems" briefly described in this section are not really problems per se, but rather properties of Unicode about which software developers should not and cannot be ignorant.

As previously mentioned, if your product must bridge multiple CJKV locales, you must deal with the problems associated with Han Unification. Specifically, you need to provide multiple glyphs per Unicode code point, as appropriate. Exactly which glyph is used for each code point depends on the locale and language. Again, please refer to Dirk Meyer's article for all of the gory details.

Unicode encoding, also known as UTF-16, can include many "null" bytes (value 0x00). All the high-frequency ASCII characters are composed of two bytes when encoded according to Unicode, one of which is the null byte. Software that has not been designed to handle UTF-16 encoding may break when encountering null bytes. UTF-8, another representation for Unicode that supports a mixed one- through six-byte encoding, was designed for use with operating systems that cannot deal with UTF-16 encoding for a variety of reasons.

Most legacy encodings are considered multiple-byte encodings, which means that the order of the bytes is the same regardless of the underlying computing architecture. UTF-16 encoding, because it is a true 16-bit encoding, can be represented in little- or big-endian byte order. Consider the Shift-JIS representation for the kanji meaning "one": 0x88EA. No matter what operating system is being used, this kanji is always encoded as 0x88EA. The UTF-16 representations for this same character are 0x4E00 (big-endian) and 0x004E (little-endian). Most UNIX and Macintosh computers are based on processors that use big-endian byte order, and most MS-DOS and Windows computers (Intel processors) use little-endian byte order.

For More Information

Unicode Version 3.0 is slated for release this year, along with an updated book. I suggest that you get a copy when it becomes available. The Unicode Consortium Web site (www.unicode.org) is also a useful resource.

Details about Unicode and its encodings, in a CJKV context, can be found in my latest book, *CJKV Information Processing* (O'Reilly, 1999). Broad coverage of legacy character sets and their encodings is also included.

Ken Lunde is manager of CJKV type development for Adobe Systems and a member of the MultiLingual Computing & Technology *Editorial Board. He can be reacbed at lunde@adobe.com*