Implementing Cross-locale CJKV Code Conversion

Ken Lunde, Adobe Systems Incorporated
lunde@adobe.com
http://www.oreilly.com/~lunde/

1. Introduction

Most operating systems today deal with single locales. Within a single CJKV locale, different operating systems often use different encodings for the same character set. Consider Shift-JIS and EUC-JP encodings for Japanese—Shift-JIS is historically used on MacOS and Windows, but EUC-JP is used on Unix. This makes code conversion a necessity. Code conversion within a single locale is, by and large, a trivial operation that typically involves a mathematical algorithm. In the past, a lot of code conversion was performed by users through dedicated software tools. Many of today's applications include built-in code conversion routines, but these routines deal with only multiple encodings of a single locale (such as EUC-KR, ISO-2022-KR, Johab, and Unified hangul Code encodings for Korean).

Code conversion across CJKV locales, such as between Chinese and Japanese, is more problematic. While Unicode serves as an excellent basis for implementing cross-locale code conversion, there are still problems to be addressed, such as unmappable characters.

2. Code Conversion Basics

Converting between different encodings of a single locale, which represents a trivial effort that involves well-established code conversion algorithms (or mapping tables), is a well-understood process these days. However, as soon as code conversion extends beyond a single locale, there are additional complexities that arise, such as the following:

- Code conversion algorithms almost always must be replaced by mapping tables because the ordering of characters in different CJKV character sets are different.
- All of the characters to be converted may not be in the target character set.

While Unicode makes for an excellent medium for cross-locale code conversion, a Unicode code point itself does not provide enough information to handle exceptional cases that are almost always guaranteed to occur when two or more locales are involved. Cross-locale code conversion requires supplemental information to increase the success rate of conversion. The pertinent issues are discussed in this paper.

Times when cross-locale code conversion is useful include when there is a document that uses characters from one locale expressed with a character set from another. For example, I sometimes receive references to Chinese documents written in Japanese. In order to render these references correctly in Chinese, I must perform cross-locale code conversion. Of course, performing this task manually is not terribly fun (though I have done it in the past), so any method for automating this is a big win in terms of efficiency and time savings.[1] Such a solution also allows people in different CJKV locales to exchange information.

3. Character Relationships

The development of Chinese characters has brought about variant forms, such as simplified and non-standard forms. Once the relationships are known, they can be expressed in a way that can be used by a code conversion program to resolve “unmappable” characters by choosing a variant.

Many characters are common to all CJKV locales, and share the same Unicode code point. These characters convert across locales with no difficulty whatsoever. Some examples are as follows:

一 山 字 人 正 大 田 白 血

The following table illustrates how some of these characters convert, trivially, from KS C 5601-1992 to JIS X 0208:1997:

Glyph	KS C 5601-1992	⟹	Unicode	⟹	JIS X 0208:1997	Glyph
一	76-73		4E00		16-76	一
山	63-03		5C71		27-19	山
田	79-03		7530		37-36	田
血	90-76		8840		23-76	血

The following sections describe some of the character relationships that need to be dealt with when implementing cross-locale code conversion in the context of Unicode. However, whenever additional information beyond the Unicode code point is used, round-trip mapping becomes less possible. There are methods for ensuring round-trip mapping if it is desired.[1]

3.1 Simplified Versus Traditional

The most well-known character relationship is that which deals with simplified and traditional characters. For every simplified character there are one or more traditional characters. However, not all character sets with simplified characters include all traditional forms. Consider the simplified kanji 黒 (JIS X 0208:1997 25-85, U+9ED2). Its traditional form 黑 is in Unicode (U+9ED1). This simplified form is used only in the Japanese locale, while its traditional form is used in the other locales. This example of a simplified/traditional pair is also Japanese-specific. Likewise, the simplified hanzi 汉 (GB 2312-80 26-26, U+6C49) is used only in the China locale, and corresponds to the Chinese character 漢 (U+6F22) in the other locales. So, GB 2312-80 26-26 first converts to U+6C49. Because U+6C49 never maps to other locales, it must be converted to U+6F22, which then maps successfully to the other locales, as follows:

CNS 11643-1992 Plane 1 73-39
JIS X 0208:1997 20-33
KS C 5601-1992 89-51

1. Wouldn’t you rather surf the web than manually converting Chinese characters from one locale to another?
1. One technique is to tag such characters with their original encoding or Unicode equivalent.

The following table illustrates what needs to take place when these two Chinese characters are converted from GB 2312-80 to JIS X 0208:1997:

Glyph	GB 2312-80	$\Longrightarrow$	Unicode	$\Longrightarrow$	Unicode'	$\Longrightarrow$	JIS X 0208:1997	Glyph
黑	26-58		9ED1		**9ED2**		25-85	黒
汉	26-26		6C49		**6F22**		20-33	漢

A large number of simplified/traditional relationships come from the China locale (the GB 2312-80 and GB/T 12345-90 character sets), and many of these relationships apply to other locales, such as Japan. For example, 国 (GB 2312-80 25-90, JIS X 0208:1997 25-81, U+56FD) is the simplified form of 國 (GB/T 12345-90 25-90, JIS X 0208:1997 52-02, U+570B) according to the China and Japan locales.

This all means that two types of simplified/traditional tables need to be built. One type is locale-independent (implemented as a single database), and the other type is locale-dependent (implemented as two or more databases, one for each supported locale).

3.2 Variants

Variant relationships are often more complex than simplified/traditional relationships because there are many types of variants. Japanese has many excellent examples of kanji with many variant forms. The following table lists several kanji with variant forms, categorized as traditional forms or general variants:

Standard Form	Traditional Forms	General Variants
学	學	斈
剣	劍	劒劎劔釼
辺	邊	邉
弁	辨瓣辯	辧

While all of these characters are included in Unicode, not all map to all CJKV locales, especially the general variants for the kanji 剣. In this case, variants in one locale that do not directly map to a character in another locale must use a variant-character database to provide a link between standard and variant forms.

3.3 Compatibility Zone Characters

Due to multiply encoded characters in some national and industry standards, such as KS C 5601-1992 and Big Five, Unicode includes a Compatibility Zone for ensuring round-trip conversion. The use of the Compatibility Zone can indeed preserve enough information to ensure round-trip conversion, but only when converting between a given locale and Unicode. The moment a second locale becomes involved, any hope for round-trip conversion is lost. These characters either become unified with *standard* Unicode characters in the process of cross-locale conversion, or else they become lost entirely.

Consider the hanja 樂 (U+6A02) which has three duplicate instances in the Compatibility Zone (U+F914, U+F95C, U+F9BF). This is because the hanja 樂 has four readings: 낙 (*nag*), 락 (*lag*), 악 (*ag*), and 요 (*yo*). In

KS C 5601-1992, this hanja is at Row-Cell 49-66, 53-05, 68-37, and 72-89. When converting to a non-Korean locale, all four instances must be unified. To add to this complexity, we can consider the case of converting these four KS C 5601-1992 characters to GB 2312-80. The GB 2312-80 equivalent, which is a simplified hanzi, is 乐 (32-54, U+4E50). The traditional form, 樂, is in GB/T 12345-90 (32-54). This means that U+6A02, U+F914, U+F95C, and U+F9BF must associate with U+4E50 in order to become the equivalent character in GB 2312-80. Consider the following conversion scenario:

Glyph	KS C 5601-1992	⟹	Unicode	⟹	Unicode'	⟹	Unicode"	⟹	GB 2312-80	Glyph
樂	49-66		F914		**6A02**		**4E50**		32-54	乐
樂	53-05		F95C		**6A02**		**4E50**		32-54	乐
樂	68-37		6A02		6A02		**4E50**		32-54	乐
樂	72-89		F9BF		**6A02**		**4E50**		32-54	乐

4. *Cross-locale Concerns*

One obvious cross-locale problem involves characters that are simply not available in the target character, not even as variant forms. Even if the definition of "variant" is stretched to the limits of the imagination, such cases do exist. The most obvious example is in Korean. Hangul cannot be converted to non-Korean locales outside the context of Unicode.

Some CJKV character sets also contain what are known as "phantom" characters. That is, characters that do not exist outside the context of the character set. The most famous phantom kanji in Japan's JIS X 0208:1997 is 彁 (55-27, U+5F41). This particular character, to put it mildly, has a snowflake's chance in hell in converting to China's GB 2312-80 or Korea's KS C 5601-1992.

Sometimes it is necessary to stretch the variant relationships a bit in order to establish a mapping for some characters. But, a distant variant of a character is almost always better than no character at all.

5. *Pitfalls*

There are many pitfalls that are difficult to avoid when performing cross-locale code conversion. While some can be treated at the character level, some are semantic in nature.

Consider the simplified hanzi 气 (GB 2312-80 38-88, U+6C14). When this hanzi is converted to JIS X 0208:1997, it becomes 气 (61-67). While the result appears to be fine, it may in fact be incorrect. The hanzi 气 is the simplified form of the hanzi 氣 (GB/T 12345-90 38-38, U+6C23). The kanji 气 in Japanese is considered a radical, not a character. If the hanzi 气 is considered a character (not a radical) in a given context, a proper mapping to Japanese would be 気 (JIS X 0208:1997 21-04, U+6C17) or 氣 (JIS X 0208:1997 61-70, U+6C23), depending on whether a simplified or traditional kanji is used. Consider the following three scenarios:

Glyph	GB 2312-80	⟹	Unicode	⟹	Unicode'	⟹	Unicode"	⟹	JIS X 0208:1997	Glyph
气	38-88		6C14						61-67	气
气	38-88		6C14		**6C23**		**6C17**		21-04	気
气	38-88		6C14		**6C23**				61-70	氣

The one that I would consider to be the best choice under most circumstances is the second one, which turns 气 into 気. If the reverse conversion were to be performed, all three JIS X 0208:1997 instances, 气, 気, and 氣, would become 气 in GB 2312-80.

6. Information Resources & Example Implementation

Performing a "diff"[1] against some Unicode mapping tables, such as for GB 2312-80 and GB/T 12345-90, can be used to generate a simple database for improving the success rate for cross-locale code conversion. Some pre-compiled databases are readily available, such as Koichi Yasuoka's "UniVariants" file.[2]

I also want to stress that using Unicode as the basis for cross-locale code conversion is beneficial because all of the databases can be expressed according to Unicode (single format). This has many advantages, perhaps the most important of which is ease of maintenance. Also consider that if there are *n* encodings, then using Unicode as the intermediate you end up needing 2*n* code conversion tables, but n^2 code conversion tables are required when converting directly from one locale to another.

I have experimented with cross-locale code conversion by building a tool written in Perl called CJKV-Conv.pl.[3] In addition to using several databases that express variant-character relationships, this tool also applies a number of multiple-byte handling techniques. All code conversion in CJKVConv.pl is done using hashes (associative arrays). The following are the hashes that are used:

- `%ihash` (key = legacy encoding; value = Unicode)
- `%ohash` (key = Unicode; value = legacy encoding)
- `%altihash` (key = legacy encoding; value = Unicode—only built if input encoding is for GB 2312-80)
- `%altohash` (key = Unicode; value = legacy encoding—only built is output encoding is for GB 2312-80)
- `%v_yasuoka` (key = Unicode; value = one or more Unicode variants derived from Koichi Yasuoka's "Univariants" database)
- `%v_jis` (key = Unicode; value = one or more Japanese-specific variants in Unicode—only accessed if input or output encodings are for JIS X 0208:1997/JIS X 0212-1990)
- `%v_cjkcompat` (key = Unicode; value = one or more mappings for "CJK Compatibility Ideographs" in Unicode)

The `%v_yasuoka`, `%v_jis`, and `%v_cjkcompat` hashes are built only if the user explicitly enables "variant" substitution using the "-v" option. The operation is quite simple. First, the input data is broken up into an

1. Unix jargon.
2. ftp://ginkaku.kudpc.kyoto-u.ac.jp/CJKtable/UniVariants.Z
3. ftp://ftp.oreilly.com/pub/examples/nutshell/ujip/perl/cjkvconv.pl

array whose elements contain one character (which may consist of one or more bytes) using the following line of code:

```
@chars = $line =~ /$lang{$il}{Enc}/gox;
```

The `$lang{$il}{Enc}` scalar variable, being used as a regular expression, contains an encoding template that defines the encoding space, which is used to break apart the input string (the contents of the `$line` scalar variable) into individual characters (the `@chars` array). A typical encoding template looks like:

```
my $euc = q{ # EUC-CN & EUC-KR encodings
  <[0-9A-F][0-9A-F][0-9A-F][0-9A-F]>      # CDPS Unicode tag
  | &U\+[0-9A-F][0-9A-F][0-9A-F][0-9A-F]; # SGML Unicode tag
  | [\x00-\x7F]                           # ASCII
  | [\xA1-\xFE][\xA1-\xFE]                # Two-byte range
};
```

Each element of the `@chars` array is then converted using the `pack()` and `unpack()` operators (because the mapping tables are expressed in hexadecimal notation for human-readability and maintenance purposes). The following represents a typical line of code used for performing code conversion using the standard `%ihash` and `%ohash` hashes:

```
$tempchar = pack("H*",$ohash{$ihash{uc unpack("H*",$char)}});
```

The code `$ihash{uc unpack("H*",$char)}` results in a four-digit hexadecimal Unicode value, which then becomes the key for `$ohash{…}`. The result is then turned back into an actual character using `pack()`. If `$tempchar` does not result in an empty string, `$char` becomes the same value, thus becoming converted:

```
$char = $tempchar;
```

All elements of the `@chars` array are finally joined so that they can be output (to standard output in this case):

```
$data = join("",@chars);
print STDOUT $data;
```

Of course, there is a lot more code in CJKVConv.pl for dealing with cases for which there is no direct mapping (using `%ihash` and `%ohash`). I encourage you to explore this program.

7. *Conclusion*

This paper has explored issues that arise when dealing with cross-locale code conversion, and provided numerous illustrative examples of so-called problematic characters from more than one locale. If we are to make cross-locale code conversion more widespread in today's applications, these issues must become understood by a wider range of developers.