Perl and Multiple-Byte Characters

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

1. Introduction

Perl currently has no built-in support for recognizing multiple-byte characters, but there is currently work underway to add Unicode support to Perl, specifically for Version 5.6.[1] (A version of Perl has been made to support particular Japanese encodings, called JPerl, but its use is not guaranteed to be portable.) There are, in the context of today's Perl, proven techniques for simulating multiple-byte support. Most of these techniques make extensive use of regular expressions, which is the context in which multiple-byte characters pose many problems.

Until Unicode support is built in, those who need to deal with multiple-byte characters in the context of multiple encodings or multiple locales should find the contents of this paper useful.

2. Multiple-byte Techniques

Typical text-processing tasks that require recognition of multiple-byte characters include searching, code conversion, filtering (that is, selective code conversion), encoding detection, and encoding integrity verification. The most important techniques for handling multiple-byte characters in these contexts include:

- Anchoring
- Trapping all characters

The following sections describe these techniques and how they are applied. But first, encoding templates need to be discussed because they are useful—in fact, critical—for both techniques.

3. The Use of Encoding Templates

Anchoring and trapping, which will be described shortly, are best implemented by using pre-defined encoding templates that specify, as a regular expression (regex), the entire theoretical encoding region for a multiple-byte encoding. Because very few encodings use all possible byte values, it is easy to apply this technique for automatically detecting encodings, and for verifying encoding integrity. The following sections provide some example encoding templates for most CJKV encoding methods. When these encoding templates are used in a regex, the `/x` modifier must be used because of the whitespace and comments used. The `/o` modifier should be used, too.

---

1. http://www.perl.com/cgi-bin/pace/pub/1999/06/perl5-6.html#unicode

An example of a mixed one- and two-byte encoding is EUC-KR. The one-byte range is 0x20 through 0x7E (we must extend this to 0x00 through 0x7F to account for control characters that may be present), and encodes ASCII or KS-Roman (KS X 1003:1993, which is equivalent to ASCII). The two-byte range is 0xA1A1 through 0xFEFE, and encodes KS X 1001:1992. If we express this in Perl as a regex tucked inside an easy-to-user scalar variable, we can get the following:

```
$euc_kr = q{ # EUC-KR encoding
    [\x00-\x7F]           # Code set 0 (ASCII or KS-Roman)
  | [\xA1-\xFE][\xA1-\xFE] # Code set 1 (KS X 1001:1992)
};
```

Liberal use of encoding templates can aid in code readability, and ease code maintenance. Many more encoding templates are provided in Section 7.

4. Anchoring

The anchoring technique works by accounting for *all* characters (made from one or more bytes, as defined in an encoding template) that lead up to a match. This technique is useful when performing regex-based searches for multiple-byte characters. The following program demonstrates this technique:

```
#!/usr/bin/perl -w

$search = "\x8C\x95";                          # "剣"
$text1  = "Text 1 \x90\x56\x8C\x95\x93\xB9"; # "Text 1 新剣道"
$text2  = "Text 2 \x94\x92\x8C\x8C\x95\x61"; # "Text 2 白血病"
$encoding = q{ # Shift-JIS encoding
    [\x00-\x7F]                               # ASCII/JIS-Roman
  | [\x81-\x9F\xE0-\xFC][\x40-\x7E\x80-\xFC] # JIS X 0208:1997
  | [\xA0-\xDF]                               # Half-width katakana
};

print "First attempt -- no anchoring\n";
print " Matched Text1\n" if $text1 =~ /$search/o;
print " Matched Text2\n" if $text2 =~ /$search/o;

print "Second attempt -- anchoring\n";
print " Matched Text1\n" if $text1 =~ /^ (?:$encoding)*? $search/osx;
print " Matched Text2\n" if $text2 =~ /^ (?:$encoding)*? $search/osx;
```

The following is the result of running the above program (assuming we name it `mb-anchor.pl`):

```
% perl mb-anchor.pl
First attempt -- no anchoring
 Matched Text1
 Matched Text2
Second attempt -- anchoring
 Matched Text1
```

Note how anchoring indeed causes correct matching to take place. The text stored in the variable `$text2` does not contain the search character (剣, 0x8C95), but its byte sequence does occur across two characters (specifically, 血 and 病, 0x8C8C and 0x9561). Without this simple recognition of multiple-byte characters, searching for multiple-byte characters can always result in false matches. Not a good thing.

But, unlike the conventional regex anchors used in Perl (such as `^` and `$`) and other regular expression implementations, these “anchors” *consume* characters.

5. Trapping All Characters

The trapping technique is useful for code conversion, filtering, encoding detection, and encoding verification. Like with the anchoring technique, this technique uses encoding templates.

5.1 Code Conversion

The following program illustrates how to break up multiple-byte data into separate array elements. This particular program doesn't do anything terribly useful, but does check whether each character consists of one or two bytes, then prints out two-byte characters along with their hexadecimal codes. The main line of code that splits the target text into individual list elements is emboldened.

```
#!/usr/bin/perl -w

$encoding = q{ # Shift-JIS encoding
    [\x00-\x7F]                                # ASCII/JIS-Roman
  | [\x81-\x9F\xE0-\xFC][\x40-\x7E\x80-\xFC] # JIS X 0208:1997
  | [\xA0-\xDF]                                # Half-width katakana
};

while (defined($line = <STDIN>)) {
  @chars = $line =~ /$encoding/gosx;      # One character per element
  foreach $char (@chars) {
    if (length($char) == 2) {             # If two-byte character
      print STDOUT "0x" . ($x = uc unpack("H*",$char), $x);
    } else {                              # All others are one-byte characters
      print STDOUT $char;
    }
  }
  print STDOUT "\n";
}
```

I find this code useful in developing code converters that make use of table-driven conversion, such as conversion between Unicode and legacy encodings. My CJKVConv.pl program, which is a Unicode-based CJKV code converter, makes use of this technique for every type of code conversion.[1] I wrote a paper for the 13th International Unicode Conference that described cross-locale code conversion techniques.[2]

5.2 Encoding Detection & Encoding Integrity Verification

A regular expression that attempts to match all characters in the target text against an encoding template can serve two useful purposes:

- Detect the encoding of the target text when multiple (and conflicting) encodings are possible—consider Shift-JIS and EUC-JP encodings for Japanese whose encoding regions overlap considerably
- Verify the integrity of the file's encoding by matching all characters against an encoding template—if even one byte cannot be accounted for, there is a possible encoding integrity problem

The following is an example of automatic code detection:

```
$is_sjs = 1 if $data =~ /^ (?:$sjs)+ $/osx;
$is_euc_jp = 1 if $data =~ /^ (?:$euc_jp)+ $/osx;
```

If variables `$sjs_out` and `$euc_out` are both true, the encoding of the target text—whether it is a single line or an entire buffer—is ambiguous. If only one is true, the target text follows that encoding or a subset

1. ftp://ftp.oreilly.com/pub/examples/nutshell/cjkv/perl/cjkvconv.pl
2. ftp://ftp.oreilly.com/pub/examples/nutshell/cjkv/unicode/iuc13-c2-paper.pdf

of each, such as pure ASCII. In fact, both `$sjs_out` and `$euc_out` would be true if the above code were applied to data consisting of only pure ASCII. Some sort of wrapper around the above code is necessary.

The following line of code can check the integrity of a file's encoding by attempting to match all bytes against an encoding template:

```
warn "Possible encoding problem at line $.!\n" if $data !~ /^ (?:$sjs)+ $/osx;
```

Consider the following program:

```
#!/usr/bin/perl -w

$sjs = q{ # Shift-JIS encoding
    [\x00-\x7F]                               # ASCII/JIS-Roman
  | [\x81-\x9F\xE0-\xFC][\x40-\x7E\x80-\xFC] # JIS X 0208:1997
  | [\xA0-\xDF]                               # Half-width katakana
};
$euc_jp = q{ # EUC-JP encoding
    [\x00-\x7F]               # Code set 0 (ASCII/JIS-Roman)
  | [\xA1-\xFE][\xA1-\xFE]     # Code set 1 (JIS X 0208:1997)
  | \x8E[\xA0-\xDF]           # Code set 2 (Half-width katakana)
  | \x8F[\xA1-\xFE][\xA1-\xFE] # Code set 3 (JIS X 0212-1990)
};

$tokyo_eucjp = "\xC5\xEC\xB5\xFE"; # 東京 ("Tokyo") in EUC-JP encoding
$tokyo_sjs   = "\x93\x8C\x8B\x9E"; # Ditto for Shift-JIS encoding

print "Attempting Shift-JIS detection\n";
print " EUC-JP text matched\n" if $tokyo_eucjp =~ /^ (?:$sjs)+ $/osx;
print " Shift-JIS text matched\n" if $tokyo_sjs =~ /^ (?:$sjs)+ $/osx;

print "Attempting EUC-JP detection\n";
print " EUC-JP text matched\n" if $tokyo_eucjp =~ /^ (?:$euc_jp)+ $/osx;
print " Shift-JIS text matched\n" if $tokyo_sjs =~ /^ (?:$euc_jp)+ $/osx;
```

The results are as follows (naming the program `detect.pl`):

```
% perl detect.pl
Attempting Shift-JIS detection
 Shift-JIS text matched
Attempting EUC-JP detection
 EUC-JP text matched
```

As expected, the same characters in different encodings are correctly detected by this short and simple program. Programming languages without regex facilities, such as C, may require several pages of code to perform the same task. The example that I used above, the two kanji 東 and 京, happen to use bytes whose values are unambiguously Shift-JIS or EUC-JP encoding. Automatic code detection may sometimes fail depending on the input. In general, the more data that you feed the above technique, the greater chances for success ("success" here is defined as matching only one encoding template, not both).

6. Algorithms or Mapping Tables?

Many techniques, such as trapping all characters, are used in CJKVConv.pl to perform code conversion and check the integrity of a file's encoding. However, code conversion usually requires huge table-driven solutions. Some types of code conversion can be effected through proven mathematical algorithms. This is usually true of multiple encoding within a single locale (such as among ISO-2022-JP, EUC-JP, and Shift-JIS encodings for Japanese) and for the various Unicode encodings (UCS-2, UCS-4, UTF-7, UTF-8, and UTF-16).

However, converting across locales or among Unicode and legacy encodings almost always requires mapping tables.

6.1 Mathematical Algorithms

The following two functions represent the algorithms for converting between ISO-2022-KR/EUC-KR and Johab encodings as far as the symbols and hanja (Chinese characters) are concerned. While some other code conversions are trivial (such as toggling a bit), the ones for dealing with Johab (Korean) and Shift-JIS (Japanese) encodings involve a lot more mathematics.

```
sub convert2johab ($) { # Convert ISO-2022-KR or EUC-KR to Johab
  my @euc = unpack("C*", $_[0]);
  my ($fe_off,$hi_off,$lo_off) = (0,0,1);
  my @out = ();

  while (($hi, $lo) = splice(@euc, 0, 2)) {
    $hi &= 127; $lo &= 127;
    $fe_off = 21 if $hi == 73;
    $fe_off = 34 if $hi == 126;
    ($hi_off,$lo_off) = ($lo_off,$hi_off) if ($hi < 74 or $hi > 125);
    push(@out, ((($hi + $hi_off) >> 1) + ($hi < 74 ? 200 : 187) - $fe_off),
      $lo + ((($hi + $lo_off) & 1) ? ($lo > 110 ? 34 : 16) : 128));
  }
  return pack("C*", @out);
}

sub johab2ks ($) { # Convert Johab to ISO-2022-KR
  my @johab = unpack("C*", $_[0]);
  my ($offset,$d8_off) = (0,0);
  my @out = ();

  while (($hi, $lo) = splice(@johab, 0, 2)) {
    $offset = 1 if ($hi > 223 and $hi < 250);
    $d8_off = ($hi == 216 and ($lo > 160 ? 94 : 42));
    push(@out, ((($hi - ($hi < 223 ? 200 : 187)) << 1) -
      ($lo < 161 ? 1 : 0) + $offset) + $d8_off),
      $lo - ($lo < 161 ? ($lo > 126 ? 34 : 16) : 128));
  }
  return pack("C*", @out);
}
```

Need I say more? One must wonder whether the mathematical complexity of the above algorithms are faster than a table-driven solution using hashes. Mathematical complexity must be compared with hash initialization and lookup.

6.2 Mapping Tables

Mapping tables, which are best implemented in Perl as hashes, can serve to support code conversion among encodings that do not have a known mathematical conversion algorithm. For example, converting among Unicode and legacy encodings requires mapping tables. Unicode can even be used to facilitate code conversion among different locales. For example, we can convert from EUC-JP (JIS X 0208:1997 and JIS X 0212-1990, Japanese) to EUC-TW (CNS 11643-1992, Traditional Chinese) encodings using two hashes: `%eucjp2uni` (EUC-JP to Unicode) and `%uni2euctw` (Unicode to EUC-TW). Consider the following program (which assumes that the hashes `%eucjp2uni` and `%uni2euctw` already exist):

```
#!/usr/bin/perl -w

$encoding = q{ # EUC-JP encoding
    [\x00-\x7F]                # Code set 0 (ASCII/JIS-Roman)
  | [\xA1-\xFE][\xA1-\xFE]     # Code set 1 (JIS X 0208:1997)
  | \x8E[\xA0-\xDF]            # Code set 2 (Half-width katakana)
  | \x8F[\xA1-\xFE][\xA1-\xFE] # Code set 3 (JIS X 0212-1990)
};

while (defined($line = <STDIN>)) {
  @chars = $line =~ /$encoding/gosx; # One character per element
  foreach $char (@chars) {
    $char = pack("H*",$uni2euctw{$eucjp2uni{unpack("H*",$char)}});
  }
  $line = join("",@chars);
  print STDOUT $line;
}
```

Note how each character is converted into a hexadecimal form using `unpack("H*",$char)`, which is then used as the key into the `%eucjp2uni` hash. The result (now in Unicode) is used as the key for the `%uni2euctw` hash. The result is packed from a hexadecimal format back into actual characters.

Of course, if the keys and values of these hashes (mapping tables) were actual multiple-byte characters instead of hexadecimal equivalents, `pack()` and `unpack()` would not be necessary. I tend to prefer hexadecimal equivalents for maintenance and human-readability reasons.

7. Encoding Template Examples

The following sections provide a number of ready-to-use CJKV encoding templates. All of these templates describe multiple-byte encodings.

7.1 Shift-JIS Encoding—Japanese

```
$sjs = q{
    [\x00-\x7F]                                # ASCII/JIS-Roman
  | [\x81-\x9F\xE0-\xFC][\x40-\x7E\x80-\xFC] # JIS X 0208:1997
  | [\xA0-\xDF]                                # Half-width katakana
};
```

7.2 EUC-JP Encoding—Japanese

```
$euc_jp = q{
    [\x00-\x7F]                # Code set 0 (ASCII/JIS-Roman)
  | [\xA1-\xFE][\xA1-\xFE]     # Code set 1 (JIS X 0208:1997)
  | \x8E[\xA0-\xDF]            # Code set 2 (Half-width katakana)
  | \x8F[\xA1-\xFE][\xA1-\xFE] # Code set 3 (JIS X 0212-1990)
};
```

7.3 Big Five Encoding—Traditional Chinese

```
$big5 = q{
    [\x00-\x7F]                    # ASCII/CNS-Roman
  | [\xA1-\xFE][\x40-\x7E\xA1-\xFE] # Big Five
};
```

7.4 GBK & Big Five Plus Encodings—Chinese

This encoding template applies equally to GBK (extended GB 2312-80) and Big Five Plus (extended Big Five) encodings.

```
$gbk = q{
    [\x00-\x7F]                     # ASCII or equivalent
  | [\x81-\xFE][\x40-\x7E\x80-\xFE] # Two-byte (GBK or Big Five Plus)
};
```

7.5 EUC-CN & EUC-KR Encodings

This encoding template applies to EUC-CN (Simplified Chinese; GB 2312-80) and EUC-KR (Korean; KS X 1001:1992) encodings.

```
$euc = q{
    [\x00-\x7F]            # Code set 0 (ASCII or equivalent)
  | [\xA1-\xFE][\xA1-\xFE] # Code set 1 (GB 2312-80 or KS X 1001:1992)
};
```

7.6 EUC-TW Encoding—Traditional Chinese

```
$euc_tw = q{
    [\x00-\x7F]                           # Code set 0 (ASCII/CNS-Roman)
  | [\xA1-\xFE][\xA1-\xFE]                # Code set 1 (CNS 11643-1992 Plane 1)
  | \x8E[\xA1-\xB0][\xA1-\xFE][\xA1-\xFE] # Code set 2 (CNS 11643-1992 Planes 1–16)
};
```

7.7 Johab Encoding—Korean

```
$johab = q{
    [\x00-\x7F]                             # ASCII/KS-Roman
  | [\x84-\xD3][\x41-\x7E\x81-\xFE]         # Modern hangul
  | [\xD8-\xDE\xE0-\xF9][\x31-\x7E\x91-\xFE] # Symbols and hanja
};
```

7.8 UCS-2 Encoding

The following encoding template works for UCS-2 encoding in any byte order (no special treatment for UTF-16 surrogates):

```
$ucs2 = q{
  [\x00-\xFF][\x00-\xFF]
};
```

Of course, the following would also work as long as the `/s` modifier is used:

```
$ucs2 = '..';
```

7.9 UTF-8 Encoding

This encoding template covers the entire UTF-8 encoding, which is a mixed one- through six-byte encoding. Note how bytes with the values 0xFE and 0xFF are explicitly excluded, and that the three- through six-byte regions explicitly exclude ranges that should never occur (to ensure that encoding integrity verification works properly).

```
$utf8 = q{
    [\x00-\x7F]                                                 # One-byte range
  | [\xC2-\xDF][\x80-\xBF]                                      # Two-byte range
  | \xE0[\xA0-\xBF][\x80-\xBF]                                  # Three-byte range
  | [\xE1-\xEF][\x80-\xBF][\x80-\xBF]                           # Three-byte range
  | \xF0[\x90-\xBF][\x80-\xBF][\x80-\xBF]                       # Four-byte range
  | [\xF1-\xF7][\x80-\xBF][\x80-\xBF][\x80-\xBF]                # Four-byte range
  | \xF8[\x88-\xBF][\x80-\xBF][\x80-\xBF][\x80-\xBF]            # Five-byte range
  | [\xF9-\xFB][\x80-\xBF][\x80-\xBF][\x80-\xBF][\x80-\xBF]     # Five-byte range
  | \xFC[\x84-\xBF][\x80-\xBF][\x80-\xBF][\x80-\xBF][\x80-\xBF] # Six-byte range
  | \xFD[\x80-\xBF][\x80-\xBF][\x80-\xBF][\x80-\xBF][\x80-\xBF] # Six-byte range
};
```

7.10 UTF-16 Encoding

This encoding template is for UTF-16 encoding in big-endian byte order with support for the surrogates area. Note that it is a mixed 16- and 32-bit (or, mixed 2- and 4-byte) encoding.

```
$utf16b = q{
    [\x00-\xD7\xE0-\xFF][\x00-\xFF]              # UCS-2
  | [\xD8-\xDB][\x00-\xFF][\xDC-\xDF][\x00-\xFF] # UTF-16 surrogates
};
```

This encoding template is for UTF-16 encoding in little-endian byte order with support for the surrogates area. Like the previous example, it is a mixed 16- and 32-bit (or, mixed 2- and 4-byte) encoding.

```
$utf16l = q{
    [\x00-\xFF][\x00-\xD7\xE0-\xFF]              # UCS-2
  | [\x00-\xFF][\xD8-\xDB][\x00-\xFF][\xDC-\xDF] # UTF-16 surrogates
};
```

8. For More Information…

Appendix W, *Perl Code Examples*, of the book *CJKV Information Processing* (O'Reilly & Associates, 1999) provides lots of Perl code that demonstrates how to manipulate multiple-byte characters. Kazumasa Utashiro's `jcode.pl` Perl library provides Japanese code conversion facilities.[1] Gisle Aas[2] and Martin Schwartz[3] have developed useful Perl modules for manipulating Unicode data and strings: `Unicode::String`, `Unicode::Map8`, and `Unicode::Map`. Also, keep a sharp eye on Unicode-related developments in Perl.[4]

1. ftp://ftp.iij.ad.jp/pub/IIJ/dist/utashiro/perl/
2. http://www.perl.com/CPAN/authors/Gisle_Aas/
3. http://www.perl.com/CPAN/authors/Martin_Schwartz/
4. Once Perl provides built-in support for Unicode, slated for Version 5.6, there is still the issue of handling legacy encodings. Legacy encodings can either be converted to Unicode (as Java does), or handled using techniques described in this paper.