Perl & Multiple-Byte Characters

Ken Lunde

Manager, CJKV Type Development

Adobe Systems Incorporated

Multiple-Byte Issues—Why Worry?

- Affects *all* CJKV (Chinese, Japanese, Korean, and Vietnamese) encodings—up to four bytes per character!
  - Locale-independent: EUC-* and ISO-2022-*
  - Locale-dependent: HZ and GBK (China), Big Five (Taiwan and Hong Kong), Shift-JIS (Japan), Johab and UHC (Korea)
- Required in the context of Unicode
  - Unicode encoding (UTF-16) is variable-length 16-bit
  - UTF-8 encoding is variable-length—one to six bytes

Remember!

One byte does not always equal one character!

Multiple-Byte Concerns

- **Code conversion**
  - For converting data into a common or different encoding
  - Required for cross-platform development
- **Data manipulation**
  - Simple text processing, such as search/replace—required for product localization
  - Related to code conversion, but not a global operation
- **Searching**
  - Simple matching—also required for product localization
- **Extensive use of regular expressions**
  - The basis for performing multiple-byte tricks through proven and related techniques such as *anchoring* and *trapping*

Code Conversion Techniques

- Algorithmic
  - — Mathematical process applied equally to *every* character
- Table-driven
  - — Requires a mapping table
  - — Round-trip *may* be an issue for unused code points
- Selective
  - — Convert only certain characters or certain character classes
  - — Can be algorithmic or table-driven—usually table-driven
  - — ***Example:*** Half- to full-width katakana—table-driven

    `ftp://ftp.oreilly.com/pub/examples/nutshell/cjkv/perl/unkana.pl`
- Combination of above techniques

Algorithmic Techniques

- **Pure algorithm**
  - Mathematical transformations are applied to *every* character
  - ***Example:*** Unicode (UTF-16) ↔ UTF-8 ↔ UTF-7
  - ***Example:*** EUC-JP ↔ ISO-2022-JP ↔ Shift-JIS
- **Partial algorithm**
  - Mathematical transformations are applied to *some* characters—table-driven for the rest
  - ***Example:*** EUC-KR ↔ ISO-2022-KR ↔ Johab
- **Zero-base—identical sequence, incompatible encoding**
  - Character codes are normalized to become a continuous sequence beginning at zero, then reset in new encoding
  - ***Example:*** EUC-CN/EUC-KR ↔ TRON Encoding

```
sub ks2tron ($) { # EUC-KR or ISO-2022-KR to TRON
  my @euc = unpack("C*", $_[0]);
  my $char;
  my @out = ();

  while (($hi, $lo) = splice(@euc, 0, 2)) {
    $hi &= 127; $lo &= 127;        # Normalize to ISO-2022-CN
    $char = (($hi - 33) * 94) + ($lo - 33); # To zero-base
    push(@out, (($char / 126) + 183), (($char % 126) + 128));
  }
  return pack("C*", @out);
}
sub gb2tron ($) { # EUC-CN or ISO-2022-CN to TRON
  my @euc = unpack("C*", $_[0]);
  my $char;
  my @out = ();

  while (($hi, $lo) = splice(@euc, 0, 2)) {
    $hi &= 127; $lo &= 127;        # Normalize to ISO-2022-KR
    $char = (($hi - 33) * 94) + ($lo - 33); # To zero-base
    push(@out, (($char / 126) + 33), (($char % 126) + 128));
  }
  return pack("C*", @out);
}
```

Table-Driven Techniques

- Used for conversion among incompatible encodings
  - — *Example:* Unicode ↔ other CJKV encodings
  - — *Example:* UHC hangul ↔ Unicode hangul
  - — *Example:* Big Five ↔ EUC-TW (CNS 11643-1992)
  - — *Example:* Half- to full-width katakana
- Hash
  - — Simple hash-based lookup using the original (unmodified) character codes
- Implemented in CJKVConv.pl:

  `ftp://ftp.oreilly.com/pub/examples/nutshell/cjkv/perl/cjkvconv.pl`

Selective Code Conversion

- Act as filters—applied to specific characters or character classes
  - — *Example:* Simplified to traditional Chinese characters

    剣→劍
  - — *Example:* Half- to full-width katakana

    ｹﾝ→ケン
- Usually table-driven, but can be semi-algorithmic

Regex Techniques

Adobe

- **Multiple-byte anchoring**
  - — Necessary for successful and correct matching of multiple-byte characters
- **Trapping *all* characters**
  - — Otherwise, matching may occur across character boundaries
- **You must specify the *complete* encoding definition in order to successfully trap or anchor multiple-byte text**
  - — Store the encoding specification trapping and anchoring
  - — Don't forget to apply the "`/ox`" regex modifiers when using the free-formatted encoding specifications that follow!
- **Encoding verification and detection**
  - — Useful in CJKV context—multiple encodings

```
$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
};

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

$utf8 = q{ # UTF-8 encoding
    [\x00-\x7F]
  | [\xC2-\xDF][\x80-\xBF]
  | \xE0[\xA0-\xBF][\x80-\xBF]
  | [\xE1-\xEF][\x80-\xBF][\x80-\xBF]
  | \xF0[\x90-\xBF][\x80-\xBF][\x80-\xBF]
  | [\xF1-\xF7][\x80-\xBF][\x80-\xBF][\x80-\xBF]
  | \xF8[\x88-\xBF][\x80-\xBF][\x80-\xBF][\x80-\xBF]
  | [\xF9-\xFB][\x80-\xBF][\x80-\xBF][\x80-\xBF][\x80-\xBF]
  | \xFC[\x84-\xBF][\x80-\xBF][\x80-\xBF][\x80-\xBF][\x80-\xBF]
  | \xFD[\x80-\xBF][\x80-\xBF][\x80-\xBF][\x80-\xBF][\x80-\xBF]
};
```

```
# Verify Shift-JIS encoding in $data:

$data =~ /^ (?:$sjs)* $/osx;

# Verify EUC-JP encoding in $data:

$data =~ /^ (?:$eucjp)* $/osx;

# Verify UTF-8 encoding in $data:

$data =~ /^ (?:$utf8)* $/osx;

# Encoding detection for data in $data:

$is_sjs   = $data =~ /^ (?:$sjs)* $/osx;
$is_eucjp = $data =~ /^ (?:$eucjp)* $/osx;

# If both $is_sjs and $is_eucjp are true (1), then the encoding
# is ambiguous. Otherwise, it is the encoding that results in
# true.
```

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

# Do searching through multiple-byte anchoring

$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]
  | [\x81-\x9F\xE0-\xFC][\x40-\x7E\x80-\xFC]
  | [\xA0-\xDF]
};

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/ox;
print " Matched Text2\n" if $text2 =~ /^(?:$encoding)*?$search/ox;
```

Regular Expression Pitfalls

- \W ("not a word") versus \w ("a word") in cross-platform environments—*you may get more than you bargained for*
  - The *standard* definition of \w:

`[0-9A-Z_a-z]` (or `[\x30-\x39\x41-\x5A\x5F\x61-\x7A]`)

  - But… MacPerl's definition of \w adds the following:

```
[\x80-\x9F\xAE\xAF\xBE\xBF\xCB-\xCF\xD8\xD9\xE5-\xEF\xF1-\xF5]
```

  - When encoding ranges are needed, use explicit ones

- **Specify the entire encoding range, including unused code points**
  - All code points from 0x00 through 0xFF must either represent themselves (that is, be one-byte characters), or else must be a valid first byte of a multiple-byte character

Advantages of Unicode

Adobe

- A *single* encoding—consider the possible encodings for the Chinese character 中 0x4E2D (“center” or “middle”)
  - Simplified Chinese = 0x5650 or 0xD6D0
  - Traditional Chinese = 0x4463, 0xC4E3, 0x8EA1C4E3, or 0xA4A4
  - Japanese = 0x4366, 0xC3E6, or 0x9286
  - Korean = 0x7169, 0xF1E9, or 0xF3E9
  - Vietnamese = 0x4A35 or 0xCAB5
- All Unicode characters—with the exception of the surrogates—are 16-bit (two bytes)
  - All characters are given equal treatment
  - No more need to deal with multiple-byte data

駱駝

Adobe