NAME

perlebcdic - Considerations for running Perl on EBCDIC platforms

DESCRIPTION

An exploration of some of the issues facing Perl programmers on EBCDIC based computers.

Portions of this document that are still incomplete are marked with XXX.

Early Perl versions worked on some EBCDIC machines, but the last known version that ran on EBCDIC was v5.8.7, until v5.22, when the Perl core again works on z/OS. Theoretically, it could work on OS/400 or Siemens' BS2000 (or their successors), but this is untested. In v5.22 and 5.24, not all the modules found on CPAN but shipped with core Perl work on z/OS.

If you want to use Perl on a non-z/OS EBCDIC machine, please let us know by sending mail to perlbug@perl.org

Writing Perl on an EBCDIC platform is really no different than writing on an ''ASCII'' one, but with different underlying numbers, as we'll see shortly. You'll have to know something about those ''ASCII'' platforms because the documentation is biased and will frequently use example numbers that don't apply to EBCDIC. There are also very few CPAN modules that are written for EBCDIC and which don't work on ASCII; instead the vast majority of CPAN modules are written for ASCII, and some may happen to work on EBCDIC, while a few have been designed to portably work on both.

If your code just uses the 52 letters A-Z and a-z, plus SPACE, the digits 0-9, and the punctuation characters that Perl uses, plus a few controls that are denoted by escape sequences like `\n` and `\t`, then there's nothing special about using Perl, and your code may very well work on an ASCII machine without change.

But if you write code that uses `\005` to mean a TAB or `\xC1` to mean an ''A'', or `\xDF` to mean a ''ÿ'' (small `"y"` with a diaeresis), then your code may well work on your EBCDIC platform, but not on an ASCII one. That's fine to do if no one will ever want to run your code on an ASCII platform; but the bias in this document will be towards writing code portable between EBCDIC and ASCII systems. Again, if every character you care about is easily enterable from your keyboard, you don't have to know anything about ASCII, but many keyboards don't easily allow you to directly enter, say, the character `\xDF`, so you have to specify it indirectly, such as by using the `"\xDF"` escape sequence. In those cases it's easiest to know something about the ASCII/Unicode character sets. If you know that the small ''ÿ'' is `U+00FF`, then you can instead specify it as `"\N{U+FF}"`, and have the computer automatically translate it to `\xDF` on your platform, and leave it as `\xFF` on ASCII ones. Or you could specify it by name, `\N{LATIN SMALL LETTER Y WITH DIAERESIS`and not ha ve to know the numbers. Either way works, but both require familiarity with Unicode.

COMMON CHARACTER CODE SETS

ASCII

The American Standard Code for Information Interchange (ASCII or US-ASCII) is a set of integers running from 0 to 127 (decimal) that have standardized interpretations by the computers which use ASCII. For example, 65 means the letter ''A''. The range 0..127 can be covered by setting various bits in a 7-bit binary digit, hence the set is sometimes referred to as ''7-bit ASCII''. ASCII was described by the American National Standards Institute document ANSI X3.4-1986. It was also described by ISO 646:1991 (with localization for currency symbols). The full ASCII set is given in the table below as the first 128 elements. Languages that can be written adequately with the characters in ASCII include English, Hawaiian, Indonesian, Swahili and some Native American languages.

Most non-EBCDIC character sets are supersets of ASCII. That is the integers 0-127 mean what ASCII says they mean. But integers 128 and above are specific to the character set.

Many of these fit entirely into 8 bits, using ASCII as 0-127, while specifying what 128-255 mean, and not using anything above 255. Thus, these are single-byte (or octet if you prefer) character sets. One important one (since Unicode is a superset of it) is the ISO 8859-1 character set.

ISO 8859

The ISO 8859-`$n` are a collection of character code sets from the International Organization for Standardization (ISO), each of which adds characters to the ASCII set that are typically found in various

languages, many of which are based on the Roman, or Latin, alphabet. Most are for European languages, but there are also ones for Arabic, Greek, Hebrew, and Thai. There are good references on the web about all these.

Latin 1 (ISO 8859−1)

A particular 8-bit extension to ASCII that includes grave and acute accented Latin characters. Languages that can employ ISO 8859-1 include all the languages covered by ASCII as well as Afrikaans, Albanian, Basque, Catalan, Danish, Faroese, Finnish, Norwegian, Portuguese, Spanish, and Swedish. Dutch is covered albeit without the ij ligature. French is covered too but without the oe ligature. German can use ISO 8859-1 but must do so without German-style quotation marks. This set is based on Western European extensions to ASCII and is commonly encountered in world wide web work. In IBM character code set identification terminology, ISO 8859-1 is also known as CCSID 819 (or sometimes 0819 or even 00819).

EBCDIC

The Extended Binary Coded Decimal Interchange Code refers to a large collection of single- and multi-byte coded character sets that are quite different from ASCII and ISO 8859-1, and are all slightly different from each other; they typically run on host computers. The EBCDIC encodings derive from 8-bit byte extensions of Hollerith punched card encodings, which long predate ASCII. The layout on the cards was such that high bits were set for the upper and lower case alphabetic characters `[a-z]` and `[A-Z]`, but there were gaps within each Latin alphabet range, visible in the table below. These gaps can cause complications.

Some IBM EBCDIC character sets may be known by character code set identification numbers (CCSID numbers) or code page numbers.

Perl can be compiled on platforms that run any of three commonly used EBCDIC character sets, listed below.

The 13 variant characters

Among IBM EBCDIC character code sets there are 13 characters that are often mapped to different integer values. Those characters are known as the 13 ‘‘variant’’ characters and are:

```
 \ [ ] { } ^ ~ ! # | $ @ `
```

When Perl is compiled for a platform, it looks at all of these characters to guess which EBCDIC character set the platform uses, and adapts itself accordingly to that platform. If the platform uses a character set that is not one of the three Perl knows about, Perl will either fail to compile, or mistakenly and silently choose one of the three.

EBCDIC code sets recognized by Perl

0037

Character code set ID 0037 is a mapping of the ASCII plus Latin-1 characters (i.e. ISO 8859-1) to an EBCDIC set. 0037 is used in North American English locales on the OS/400 operating system that runs on AS/400 computers. CCSID 0037 differs from ISO 8859-1 in 236 places; in other words they agree on only 20 code point values.

1047

Character code set ID 1047 is also a mapping of the ASCII plus Latin-1 characters (i.e. ISO 8859-1) to an EBCDIC set. 1047 is used under Unix System Services for OS/390 or z/OS, and OpenEdition for VM/ESA. CCSID 1047 differs from CCSID 0037 in eight places, and from ISO 8859-1 in 236.

POSIX-BC

The EBCDIC code page in use on Siemens’ BS2000 system is distinct from 1047 and 0037. It is identified below as the POSIX-BC set. Like 0037 and 1047, it is the same as ISO 8859-1 in 20 code point values.

Unicode code points versus EBCDIC code points

In Unicode terminology a *code point* is the number assigned to a character: for example, in EBCDIC the character ‘‘A’’ is usually assigned the number 193. In Unicode, the character ‘‘A’’ is assigned the number 65. All the code points in ASCII and Latin-1 (ISO 8859-1) have the same meaning in Unicode. All three of

the recognized EBCDIC code sets have 256 code points, and in each code set, all 256 code points are mapped to equivalent Latin1 code points. Obviously, ‘‘A’’ will map to ‘‘A’’, ‘‘B’’ => ‘‘B’’, ‘‘%’’ => ‘‘%’’, etc., for all printable characters in Latin1 and these code pages.

It also turns out that EBCDIC has nearly precise equivalents for the ASCII/Latin1 C0 controls and the DELETE control. (The C0 controls are those whose ASCII code points are 0..0x1F; things like TAB, ACK, BEL, etc.) A mapping is set up between these ASCII/EBCDIC controls. There isn’t such a precise mapping between the C1 controls on ASCII platforms and the remaining EBCDIC controls. What has been done is to map these controls, mostly arbitrarily, to some otherwise unmatched character in the other character set. Most of these are very very rarely used nowadays in EBCDIC anyway, and their names have been dropped, without much complaint. For example the EO (Eight Ones) EBCDIC control (consisting of eight one bits = 0xFF) is mapped to the C1 APC control (0x9F), and you can’t use the name ‘‘EO’’.

The EBCDIC controls provide three possible line terminator characters, CR (0x0D), LF (0x25), and NL (0x15). On ASCII platforms, the symbols ‘‘NL’’ and ‘‘LF’’ refer to the same character, but in strict EBCDIC terminology they are different ones. The EBCDIC NL is mapped to the C1 control called ‘‘NEL’’ (‘‘Next Line’’; here’s a case where the mapping makes quite a bit of sense, and hence isn’t just arbitrary). On some EBCDIC platforms, this NL or NEL is the typical line terminator. This is true of z/OS and BS2000. In these platforms, the C compilers will swap the LF and NEL code points, so that `"\n"` is 0x15, and refers to NL. Perl does that too; you can see it in the code chart below. This makes things generally ‘‘just work’’ without you even having to be aware that there is a swap.

Unicode and UTF

UTF stands for ‘‘Unicode Transformation Format’’. UTF-8 is an encoding of Unicode into a sequence of 8-bit byte chunks, based on ASCII and Latin-1. The length of a sequence required to represent a Unicode code point depends on the ordinal number of that code point, with larger numbers requiring more bytes. UTF-EBCDIC is like UTF-8, but based on EBCDIC. They are enough alike that often, casual usage will conflate the two terms, and use ‘‘UTF-8’’ to mean both the UTF-8 found on ASCII platforms, and the UTF-EBCDIC found on EBCDIC ones.

You may see the term ‘‘invariant’’ character or code point. This simply means that the character has the same numeric value and representation when encoded in UTF-8 (or UTF-EBCDIC) as when not. (Note that this is a very different concept from ‘‘The 13 variant characters’’ mentioned above. Careful prose will use the term ‘‘UTF-8 invariant’’ instead of just ‘‘invariant’’, but most often you’ll see just ‘‘invariant’’.) For example, the ordinal value of ‘‘A’’ is 193 in most EBCDIC code pages, and also is 193 when encoded in UTF-EBCDIC. All UTF-8 (or UTF-EBCDIC) variant code points occupy at least two bytes when encoded in UTF-8 (or UTF-EBCDIC); by definition, the UTF-8 (or UTF-EBCDIC) invariant code points are exactly one byte whether encoded in UTF-8 (or UTF-EBCDIC), or not. (By now you see why people typically just say ‘‘UTF-8’’ when they also mean ‘‘UTF-EBCDIC’’. For the rest of this document, we’ll mostly be casual about it too.) In ASCII UTF-8, the code points corresponding to the lowest 128 ordinal numbers (0 - 127: the ASCII characters) are invariant. In UTF-EBCDIC, there are 160 invariant characters. (If you care, the EBCDIC invariants are those characters which have ASCII equivalents, plus those that correspond to the C1 controls (128 - 159 on ASCII platforms).)

A string encoded in UTF-EBCDIC may be longer (very rarely shorter) than one encoded in UTF-8. Perl extends both UTF-8 and UTF-EBCDIC so that they can encode code points above the Unicode maximum of U+10FFFF. Both extensions are constructed to allow encoding of any code point that fits in a 64-bit word.

UTF-EBCDIC is defined by Unicode Technical Report #16 <http://www.unicode.org/reports/tr16> (often referred to as just TR16). It is defined based on CCSID 1047, not allowing for the differences for other code pages. This allows for easy interchange of text between computers running different code pages, but makes it unusable, without adaptation, for Perl on those other code pages.

The reason for this unusability is that a fundamental assumption of Perl is that the characters it cares about for parsing and lexical analysis are the same whether or not the text is in UTF-8. For example, Perl expects the character `"["` to have the same representation, no matter if the string containing it (or program text) is UTF-8 encoded or not. To ensure this, Perl adapts UTF-EBCDIC to the particular code page so that all characters it expects to be UTF-8 invariant are in fact UTF-8 invariant. This means that text generated on a computer running one version of Perl’s UTF-EBCDIC has to be translated to be intelligible to a computer

running another.

TR16 implies a method to extend UTF-EBCDIC to encode points up through `2**31-1`. Perl uses this method for code points up through `2**30-1`, but uses an incompatible method for larger ones, to enable it to handle much larger code points than otherwise.

Using Encode

Starting from Perl 5.8 you can use the standard module Encode to translate from EBCDIC to Latin-1 code points. Encode knows about more EBCDIC character sets than Perl can currently be compiled to run on.

```
use Encode 'from_to';

my %ebcdic = ( 176 => 'cp37', 95 => 'cp1047', 106 => 'posix-bc' );

# $a is in EBCDIC code points
from_to($a, $ebcdic{ord '^'}, 'latin1');
# $a is ISO 8859-1 code points
```

and from Latin-1 code points to EBCDIC code points

```
use Encode 'from_to';

my %ebcdic = ( 176 => 'cp37', 95 => 'cp1047', 106 => 'posix-bc' );

# $a is ISO 8859-1 code points
from_to($a, 'latin1', $ebcdic{ord '^'});
# $a is in EBCDIC code points
```

For doing I/O it is suggested that you use the autotranslating features of PerlIO, see perluniintro.

Since version 5.8 Perl uses the PerlIO I/O library. This enables you to use different encodings per IO channel. For example you may use

```
use Encode;
open($f, ">:encoding(ascii)", "test.ascii");
print $f "Hello World!\n";
open($f, ">:encoding(cp37)", "test.ebcdic");
print $f "Hello World!\n";
open($f, ">:encoding(latin1)", "test.latin1");
print $f "Hello World!\n";
open($f, ">:encoding(utf8)", "test.utf8");
print $f "Hello World!\n";
```

to get four files containing "Hello World!\n" in ASCII, CP 0037 EBCDIC, ISO 8859-1 (Latin-1) (in this example identical to ASCII since only ASCII characters were printed), and UTF-EBCDIC (in this example identical to normal EBCDIC since only characters that don't differ between EBCDIC and UTF-EBCDIC were printed). See the documentation of Encode::PerlIO for details.

As the PerlIO layer uses raw IO (bytes) internally, all this totally ignores things like the type of your filesystem (ASCII or EBCDIC).

SINGLE OCTET TABLES

The following tables list the ASCII and Latin 1 ordered sets including the subsets: C0 controls (0..31), ASCII graphics (32..7e), delete (7f), C1 controls (80..9f), and Latin-1 (a.k.a. ISO 8859-1) (a0..ff). In the table names of the Latin 1 extensions to ASCII have been labelled with character names roughly corresponding to *The Unicode Standard, Version 6.1* albeit with substitutions such as `s/LATIN//` and `s/VULGAR//` in all cases; `s/CAPITALLETTER//` in some cases; and `s/SMALLLETTER([A-Z])/\l$1/` in some other cases. Controls are listed using their Unicode 6.2 abbreviations. The differences between the 0037 and 1047 sets are flagged with `**`. The differences between the 1047 and POSIX-BC sets are flagged with `##`. All `ord()` numbers listed are decimal. If you would rather see this table listing octal values, then run the table (that is, the pod source text of this document, since this recipe may not work with a

pod2_other_format translation) through:

recipe 0

```
perl -ne 'if(/(.{29})(\d+)\s+(\d+)\s+(\d+)\s+(\d+)/)' \
-e '{printf("%s%-5.03o%-5.03o%-5.03o%.03o\n",$1,$2,$3,$4,$5)}' \
perlebcdic.pod
```

If you want to retain the UTF-x code points then in script form you might want to write:

recipe 1

```
open(FH,"<perlebcdic.pod") or die "Could not open perlebcdic.pod: $!";
while (<FH>) {
if (/(.{29})(\d+)\s+(\d+)\s+(\d+)\s+(\d+)\s+(\d+)\.?(\d*)
\s+(\d+)\.?(\d*)/x)
{
if ($7 ne '' && $9 ne '') {
printf(
"%s%-5.03o%-5.03o%-5.03o%-5.03o%-3o.%-5o%-3o.%.03o\n",
$1,$2,$3,$4,$5,$6,$7,$8,$9);
}
elsif ($7 ne '') {
printf("%s%-5.03o%-5.03o%-5.03o%-5.03o%-3o.%-5o%.03o\n",
$1,$2,$3,$4,$5,$6,$7,$8);
}
else {
printf("%s%-5.03o%-5.03o%-5.03o%-5.03o%-5.03o%.03o\n",
$1,$2,$3,$4,$5,$6,$8);
}
}
}
```

If you would rather see this table listing hexadecimal values then run the table through:

recipe 2

```
perl -ne 'if(/(.{29})(\d+)\s+(\d+)\s+(\d+)\s+(\d+)/)' \
-e '{printf("%s%-5.02X%-5.02X%-5.02X%.02X\n",$1,$2,$3,$4,$5)}' \
perlebcdic.pod
```

Or, in order to retain the UTF-x code points in hexadecimal:

recipe 3

```
open(FH,"<perlebcdic.pod") or die "Could not open perlebcdic.pod: $!";
while (<FH>) {
if (/(.{29})(\d+)\s+(\d+)\s+(\d+)\s+(\d+)\s+(\d+)\.?(\d*)
\s+(\d+)\.?(\d*)/x)
{
if ($7 ne '' && $9 ne '') {
printf(
"%s%-5.02X%-5.02X%-5.02X%-5.02X%-2X.%-6.02X%02X.%02X\n",
$1,$2,$3,$4,$5,$6,$7,$8,$9);
}
elsif ($7 ne '') {
printf("%s%-5.02X%-5.02X%-5.02X%-5.02X%-2X.%-6.02X%02X\n",
$1,$2,$3,$4,$5,$6,$7,$8);
}
else {
printf("%s%-5.02X%-5.02X%-5.02X%-5.02X%-5.02X%02X\n",
```

```
$1,$2,$3,$4,$5,$6,$8);
}
}
}
```

```
ISO
8859-1 POS- CCSID
CCSID CCSID CCSID IX- 1047
chr 0819 0037 1047 BC UTF-8 UTF-EBCDIC
---------------------------------------------------------------------
<NUL> 0 0 0 0 0 0
<SOH> 1 1 1 1 1 1
<STX> 2 2 2 2 2 2
<ETX> 3 3 3 3 3 3
<EOT> 4 55 55 55 4 55
<ENQ> 5 45 45 45 5 45
<ACK> 6 46 46 46 6 46
<BEL> 7 47 47 47 7 47
<BS> 8 22 22 22 8 22
<HT> 9 5 5 5 9 5
<LF> 10 37 21 21 10 21 **
<VT> 11 11 11 11 11 11
<FF> 12 12 12 12 12 12
<CR> 13 13 13 13 13 13
<SO> 14 14 14 14 14 14
<SI> 15 15 15 15 15 15
<DLE> 16 16 16 16 16 16
<DC1> 17 17 17 17 17 17
<DC2> 18 18 18 18 18 18
<DC3> 19 19 19 19 19 19
<DC4> 20 60 60 60 20 60
<NAK> 21 61 61 61 21 61
<SYN> 22 50 50 50 22 50
<ETB> 23 38 38 38 23 38
<CAN> 24 24 24 24 24 24
<EOM> 25 25 25 25 25 25
<SUB> 26 63 63 63 26 63
<ESC> 27 39 39 39 27 39
<FS> 28 28 28 28 28 28
<GS> 29 29 29 29 29 29
<RS> 30 30 30 30 30 30
<US> 31 31 31 31 31 31
<SPACE> 32 64 64 64 32 64
! 33 90 90 90 33 90
" 34 127 127 127 34 127
# 35 123 123 123 35 123
$ 36 91 91 91 36 91
% 37 108 108 108 37 108
& 38 80 80 80 38 80
' 39 125 125 125 39 125
( 40 77 77 77 40 77
) 41 93 93 93 41 93
* 42 92 92 92 42 92
```

```
+ 43 78 78 78 43 78
, 44 107 107 107 44 107
- 45 96 96 96 45 96
. 46 75 75 75 46 75
/ 47 97 97 97 47 97
0 48 240 240 240 48 240
1 49 241 241 241 49 241
2 50 242 242 242 50 242
3 51 243 243 243 51 243
4 52 244 244 244 52 244
5 53 245 245 245 53 245
6 54 246 246 246 54 246
7 55 247 247 247 55 247
8 56 248 248 248 56 248
9 57 249 249 249 57 249
: 58 122 122 122 58 122
; 59 94 94 94 59 94
< 60 76 76 76 60 76
= 61 126 126 126 61 126
> 62 110 110 110 62 110
? 63 111 111 111 63 111
@ 64 124 124 124 64 124
A 65 193 193 193 65 193
B 66 194 194 194 66 194
C 67 195 195 195 67 195
D 68 196 196 196 68 196
E 69 197 197 197 69 197
F 70 198 198 198 70 198
G 71 199 199 199 71 199
H 72 200 200 200 72 200
I 73 201 201 201 73 201
J 74 209 209 209 74 209
K 75 210 210 210 75 210
L 76 211 211 211 76 211
M 77 212 212 212 77 212
N 78 213 213 213 78 213
O 79 214 214 214 79 214
P 80 215 215 215 80 215
Q 81 216 216 216 81 216
R 82 217 217 217 82 217
S 83 226 226 226 83 226
T 84 227 227 227 84 227
U 85 228 228 228 85 228
V 86 229 229 229 86 229
W 87 230 230 230 87 230
X 88 231 231 231 88 231
Y 89 232 232 232 89 232
Z 90 233 233 233 90 233
[ 91 186 173 187 91 173 ** ##
\ 92 224 224 188 92 224 ##
] 93 187 189 189 93 189 **
^ 94 176 95 106 94 95 ** ##
_ 95 109 109 109 95 109
` 96 121 121 74 96 121 ##
```

```
a 97 129 129 129 97 129
b 98 130 130 130 98 130
c 99 131 131 131 99 131
d 100 132 132 132 100 132
e 101 133 133 133 101 133
f 102 134 134 134 102 134
g 103 135 135 135 103 135
h 104 136 136 136 104 136
i 105 137 137 137 105 137
j 106 145 145 145 106 145
k 107 146 146 146 107 146
l 108 147 147 147 108 147
m 109 148 148 148 109 148
n 110 149 149 149 110 149
o 111 150 150 150 111 150
p 112 151 151 151 112 151
q 113 152 152 152 113 152
r 114 153 153 153 114 153
s 115 162 162 162 115 162
t 116 163 163 163 116 163
u 117 164 164 164 117 164
v 118 165 165 165 118 165
w 119 166 166 166 119 166
x 120 167 167 167 120 167
y 121 168 168 168 121 168
z 122 169 169 169 122 169
{ 123 192 192 251 123 192 ##
| 124 79 79 79 124 79
} 125 208 208 253 125 208 ##
~ 126 161 161 255 126 161 ##
<DEL> 127 7 7 7 127 7
<PAD> 128 32 32 32 194.128 32
<HOP> 129 33 33 33 194.129 33
<BPH> 130 34 34 34 194.130 34
<NBH> 131 35 35 35 194.131 35
<IND> 132 36 36 36 194.132 36
<NEL> 133 21 37 37 194.133 37 **
<SSA> 134 6 6 6 194.134 6
<ESA> 135 23 23 23 194.135 23
<HTS> 136 40 40 40 194.136 40
<HTJ> 137 41 41 41 194.137 41
<VTS> 138 42 42 42 194.138 42
<PLD> 139 43 43 43 194.139 43
<PLU> 140 44 44 44 194.140 44
<RI> 141 9 9 9 194.141 9
<SS2> 142 10 10 10 194.142 10
<SS3> 143 27 27 27 194.143 27
<DCS> 144 48 48 48 194.144 48
<PU1> 145 49 49 49 194.145 49
<PU2> 146 26 26 26 194.146 26
<STS> 147 51 51 51 194.147 51
<CCH> 148 52 52 52 194.148 52
<MW> 149 53 53 53 194.149 53
<SPA> 150 54 54 54 194.150 54
```

```
<EPA> 151 8 8 8 194.151 8
<SOS> 152 56 56 56 194.152 56
<SGC> 153 57 57 57 194.153 57
<SCI> 154 58 58 58 194.154 58
<CSI> 155 59 59 59 194.155 59
<ST> 156 4 4 4 194.156 4
<OSC> 157 20 20 20 194.157 20
<PM> 158 62 62 62 194.158 62
<APC> 159 255 255 95 194.159 255 ##
<NON-BREAKING SPACE> 160 65 65 65 194.160 128.65
<INVERTED "!" > 161 170 170 170 194.161 128.66
<CENT SIGN> 162 74 74 176 194.162 128.67 ##
<POUND SIGN> 163 177 177 177 194.163 128.68
<CURRENCY SIGN> 164 159 159 159 194.164 128.69
<YEN SIGN> 165 178 178 178 194.165 128.70
<BROKEN BAR> 166 106 106 208 194.166 128.71 ##
<SECTION SIGN> 167 181 181 181 194.167 128.72
<DIAERESIS> 168 189 187 121 194.168 128.73 ** ##
<COPYRIGHT SIGN> 169 180 180 180 194.169 128.74
<FEMININE ORDINAL> 170 154 154 154 194.170 128.81
<LEFT POINTING GUILLEMET> 171 138 138 138 194.171 128.82
<NOT SIGN> 172 95 176 186 194.172 128.83 ** ##
<SOFT HYPHEN> 173 202 202 202 194.173 128.84
<REGISTERED TRADE MARK> 174 175 175 175 194.174 128.85
<MACRON> 175 188 188 161 194.175 128.86 ##
<DEGREE SIGN> 176 144 144 144 194.176 128.87
<PLUS-OR-MINUS SIGN> 177 143 143 143 194.177 128.88
<SUPERSCRIPT TWO> 178 234 234 234 194.178 128.89
<SUPERSCRIPT THREE> 179 250 250 250 194.179 128.98
<ACUTE ACCENT> 180 190 190 190 194.180 128.99
<MICRO SIGN> 181 160 160 160 194.181 128.100
<PARAGRAPH SIGN> 182 182 182 182 194.182 128.101
<MIDDLE DOT> 183 179 179 179 194.183 128.102
<CEDILLA> 184 157 157 157 194.184 128.103
<SUPERSCRIPT ONE> 185 218 218 218 194.185 128.104
<MASC. ORDINAL INDICATOR> 186 155 155 155 194.186 128.105
<RIGHT POINTING GUILLEMET> 187 139 139 139 194.187 128.106
<FRACTION ONE QUARTER> 188 183 183 183 194.188 128.112
<FRACTION ONE HALF> 189 184 184 184 194.189 128.113
<FRACTION THREE QUARTERS> 190 185 185 185 194.190 128.114
<INVERTED QUESTION MARK> 191 171 171 171 194.191 128.115
<A WITH GRAVE> 192 100 100 100 195.128 138.65
<A WITH ACUTE> 193 101 101 101 195.129 138.66
<A WITH CIRCUMFLEX> 194 98 98 98 195.130 138.67
<A WITH TILDE> 195 102 102 102 195.131 138.68
<A WITH DIAERESIS> 196 99 99 99 195.132 138.69
<A WITH RING ABOVE> 197 103 103 103 195.133 138.70
<CAPITAL LIGATURE AE> 198 158 158 158 195.134 138.71
<C WITH CEDILLA> 199 104 104 104 195.135 138.72
<E WITH GRAVE> 200 116 116 116 195.136 138.73
<E WITH ACUTE> 201 113 113 113 195.137 138.74
<E WITH CIRCUMFLEX> 202 114 114 114 195.138 138.81
<E WITH DIAERESIS> 203 115 115 115 195.139 138.82
<I WITH GRAVE> 204 120 120 120 195.140 138.83
```

```
<I WITH ACUTE> 205 117 117 117 195.141 138.84
<I WITH CIRCUMFLEX> 206 118 118 118 195.142 138.85
<I WITH DIAERESIS> 207 119 119 119 195.143 138.86
<CAPITAL LETTER ETH> 208 172 172 172 195.144 138.87
<N WITH TILDE> 209 105 105 105 195.145 138.88
<O WITH GRAVE> 210 237 237 237 195.146 138.89
<O WITH ACUTE> 211 238 238 238 195.147 138.98
<O WITH CIRCUMFLEX> 212 235 235 235 195.148 138.99
<O WITH TILDE> 213 239 239 239 195.149 138.100
<O WITH DIAERESIS> 214 236 236 236 195.150 138.101
<MULTIPLICATION SIGN> 215 191 191 191 195.151 138.102
<O WITH STROKE> 216 128 128 128 195.152 138.103
<U WITH GRAVE> 217 253 253 224 195.153 138.104 ##
<U WITH ACUTE> 218 254 254 254 195.154 138.105
<U WITH CIRCUMFLEX> 219 251 251 221 195.155 138.106 ##
<U WITH DIAERESIS> 220 252 252 252 195.156 138.112
<Y WITH ACUTE> 221 173 186 173 195.157 138.113 ** ##
<CAPITAL LETTER THORN> 222 174 174 174 195.158 138.114
<SMALL LETTER SHARP S> 223 89 89 89 195.159 138.115
<a WITH GRAVE> 224 68 68 68 195.160 139.65
<a WITH ACUTE> 225 69 69 69 195.161 139.66
<a WITH CIRCUMFLEX> 226 66 66 66 195.162 139.67
<a WITH TILDE> 227 70 70 70 195.163 139.68
<a WITH DIAERESIS> 228 67 67 67 195.164 139.69
<a WITH RING ABOVE> 229 71 71 71 195.165 139.70
<SMALL LIGATURE ae> 230 156 156 156 195.166 139.71
<c WITH CEDILLA> 231 72 72 72 195.167 139.72
<e WITH GRAVE> 232 84 84 84 195.168 139.73
<e WITH ACUTE> 233 81 81 81 195.169 139.74
<e WITH CIRCUMFLEX> 234 82 82 82 195.170 139.81
<e WITH DIAERESIS> 235 83 83 83 195.171 139.82
<i WITH GRAVE> 236 88 88 88 195.172 139.83
<i WITH ACUTE> 237 85 85 85 195.173 139.84
<i WITH CIRCUMFLEX> 238 86 86 86 195.174 139.85
<i WITH DIAERESIS> 239 87 87 87 195.175 139.86
<SMALL LETTER eth> 240 140 140 140 195.176 139.87
<n WITH TILDE> 241 73 73 73 195.177 139.88
<o WITH GRAVE> 242 205 205 205 195.178 139.89
<o WITH ACUTE> 243 206 206 206 195.179 139.98
<o WITH CIRCUMFLEX> 244 203 203 203 195.180 139.99
<o WITH TILDE> 245 207 207 207 195.181 139.100
<o WITH DIAERESIS> 246 204 204 204 195.182 139.101
<DIVISION SIGN> 247 225 225 225 195.183 139.102
<o WITH STROKE> 248 112 112 112 195.184 139.103
<u WITH GRAVE> 249 221 221 192 195.185 139.104 ##
<u WITH ACUTE> 250 222 222 222 195.186 139.105
<u WITH CIRCUMFLEX> 251 219 219 219 195.187 139.106
<u WITH DIAERESIS> 252 220 220 220 195.188 139.112
<y WITH ACUTE> 253 141 141 141 195.189 139.113
<SMALL LETTER thorn> 254 142 142 142 195.190 139.114
<y WITH DIAERESIS> 255 223 223 223 195.191 139.115
```

If you would rather see the above table in CCSID 0037 order rather than ASCII + Latin-1 order then run the table through:

recipe 4

```
perl \
-ne 'if(/.{29}\d{1,3}\s{2,4}\d{1,3}\s{2,4}\d{1,3}\s{2,4}\d{1,3}/)'\
-e '{push(@l,$_)}' \
-e 'END{print map{$_->[0]}' \
-e ' sort{$a->[1] <=> $b->[1]}' \
-e ' map{[$_,substr($_,34,3)]}@l;}' perlebcdic.pod
```

If you would rather see it in CCSID 1047 order then change the number 34 in the last line to 39, like this:

recipe 5

```
perl \
-ne 'if(/.{29}\d{1,3}\s{2,4}\d{1,3}\s{2,4}\d{1,3}\s{2,4}\d{1,3}/)'\
-e '{push(@l,$_)}' \
-e 'END{print map{$_->[0]}' \
-e ' sort{$a->[1] <=> $b->[1]}' \
-e ' map{[$_,substr($_,39,3)]}@l;}' perlebcdic.pod
```

If you would rather see it in POSIX-BC order then change the number 34 in the last line to 44, like this:

recipe 6

```
perl \
-ne 'if(/.{29}\d{1,3}\s{2,4}\d{1,3}\s{2,4}\d{1,3}\s{2,4}\d{1,3}/)'\
-e '{push(@l,$_)}' \
-e 'END{print map{$_->[0]}' \
-e ' sort{$a->[1] <=> $b->[1]}' \
-e ' map{[$_,substr($_,44,3)]}@l;}' perlebcdic.pod
```

Table in hex, sorted in 1047 order

Since this document was first written, the convention has become more and more to use hexadecimal notation for code points. To do this with the recipes and to also sort is a multi-step process, so here, for convenience, is the table from above, re-sorted to be in Code Page 1047 order, and using hex notation.

```
ISO
8859-1 POS- CCSID
CCSID CCSID CCSID IX- 1047
chr 0819 0037 1047 BC UTF-8 UTF-EBCDIC
-----------------------------------------------------------------------
<NUL> 00 00 00 00 00 00
<SOH> 01 01 01 01 01 01
<STX> 02 02 02 02 02 02
<ETX> 03 03 03 03 03 03
<ST> 9C 04 04 04 C2.9C 04
<HT> 09 05 05 05 09 05
<SSA> 86 06 06 06 C2.86 06
<DEL> 7F 07 07 07 7F 07
<EPA> 97 08 08 08 C2.97 08
<RI> 8D 09 09 09 C2.8D 09
<SS2> 8E 0A 0A 0A C2.8E 0A
<VT> 0B 0B 0B 0B 0B 0B
<FF> 0C 0C 0C 0C 0C 0C
<CR> 0D 0D 0D 0D 0D 0D
<SO> 0E 0E 0E 0E 0E 0E
<SI> 0F 0F 0F 0F 0F 0F
<DLE> 10 10 10 10 10 10
<DC1> 11 11 11 11 11 11
<DC2> 12 12 12 12 12 12
```

```
<DC3> 13 13 13 13 13 13
<OSC> 9D 14 14 14 C2.9D 14
<LF> 0A 25 15 15 0A 15 **
<BS> 08 16 16 16 08 16
<ESA> 87 17 17 17 C2.87 17
<CAN> 18 18 18 18 18 18
<EOM> 19 19 19 19 19 19
<PU2> 92 1A 1A 1A C2.92 1A
<SS3> 8F 1B 1B 1B C2.8F 1B
<FS> 1C 1C 1C 1C 1C 1C
<GS> 1D 1D 1D 1D 1D 1D
<RS> 1E 1E 1E 1E 1E 1E
<US> 1F 1F 1F 1F 1F 1F
<PAD> 80 20 20 20 C2.80 20
<HOP> 81 21 21 21 C2.81 21
<BPH> 82 22 22 22 C2.82 22
<NBH> 83 23 23 23 C2.83 23
<IND> 84 24 24 24 C2.84 24
<NEL> 85 15 25 25 C2.85 25 **
<ETB> 17 26 26 26 17 26
<ESC> 1B 27 27 27 1B 27
<HTS> 88 28 28 28 C2.88 28
<HTJ> 89 29 29 29 C2.89 29
<VTS> 8A 2A 2A 2A C2.8A 2A
<PLD> 8B 2B 2B 2B C2.8B 2B
<PLU> 8C 2C 2C 2C C2.8C 2C
<ENQ> 05 2D 2D 2D 05 2D
<ACK> 06 2E 2E 2E 06 2E
<BEL> 07 2F 2F 2F 07 2F
<DCS> 90 30 30 30 C2.90 30
<PU1> 91 31 31 31 C2.91 31
<SYN> 16 32 32 32 16 32
<STS> 93 33 33 33 C2.93 33
<CCH> 94 34 34 34 C2.94 34
<MW> 95 35 35 35 C2.95 35
<SPA> 96 36 36 36 C2.96 36
<EOT> 04 37 37 37 04 37
<SOS> 98 38 38 38 C2.98 38
<SGC> 99 39 39 39 C2.99 39
<SCI> 9A 3A 3A 3A C2.9A 3A
<CSI> 9B 3B 3B 3B C2.9B 3B
<DC4> 14 3C 3C 3C 14 3C
<NAK> 15 3D 3D 3D 15 3D
<PM> 9E 3E 3E 3E C2.9E 3E
<SUB> 1A 3F 3F 3F 1A 3F
<SPACE> 20 40 40 40 20 40
<NON-BREAKING SPACE> A0 41 41 41 C2.A0 80.41
<a WITH CIRCUMFLEX> E2 42 42 42 C3.A2 8B.43
<a WITH DIAERESIS> E4 43 43 43 C3.A4 8B.45
<a WITH GRAVE> E0 44 44 44 C3.A0 8B.41
<a WITH ACUTE> E1 45 45 45 C3.A1 8B.42
<a WITH TILDE> E3 46 46 46 C3.A3 8B.44
<a WITH RING ABOVE> E5 47 47 47 C3.A5 8B.46
<c WITH CEDILLA> E7 48 48 48 C3.A7 8B.48
```

```
<n WITH TILDE> F1 49 49 49 C3.B1 8B.58
<CENT SIGN> A2 4A 4A B0 C2.A2 80.43 ##
. 2E 4B 4B 4B 2E 4B
< 3C 4C 4C 4C 3C 4C
( 28 4D 4D 4D 28 4D
+ 2B 4E 4E 4E 2B 4E
| 7C 4F 4F 4F 7C 4F
& 26 50 50 50 26 50
<e WITH ACUTE> E9 51 51 51 C3.A9 8B.4A
<e WITH CIRCUMFLEX> EA 52 52 52 C3.AA 8B.51
<e WITH DIAERESIS> EB 53 53 53 C3.AB 8B.52
<e WITH GRAVE> E8 54 54 54 C3.A8 8B.49
<i WITH ACUTE> ED 55 55 55 C3.AD 8B.54
<i WITH CIRCUMFLEX> EE 56 56 56 C3.AE 8B.55
<i WITH DIAERESIS> EF 57 57 57 C3.AF 8B.56
<i WITH GRAVE> EC 58 58 58 C3.AC 8B.53
<SMALL LETTER SHARP S> DF 59 59 59 C3.9F 8A.73
! 21 5A 5A 5A 21 5A
$ 24 5B 5B 5B 24 5B
* 2A 5C 5C 5C 2A 5C
) 29 5D 5D 5D 29 5D
; 3B 5E 5E 5E 3B 5E
^ 5E B0 5F 6A 5E 5F ** ##
- 2D 60 60 60 2D 60
/ 2F 61 61 61 2F 61
<A WITH CIRCUMFLEX> C2 62 62 62 C3.82 8A.43
<A WITH DIAERESIS> C4 63 63 63 C3.84 8A.45
<A WITH GRAVE> C0 64 64 64 C3.80 8A.41
<A WITH ACUTE> C1 65 65 65 C3.81 8A.42
<A WITH TILDE> C3 66 66 66 C3.83 8A.44
<A WITH RING ABOVE> C5 67 67 67 C3.85 8A.46
<C WITH CEDILLA> C7 68 68 68 C3.87 8A.48
<N WITH TILDE> D1 69 69 69 C3.91 8A.58
<BROKEN BAR> A6 6A 6A D0 C2.A6 80.47 ##
, 2C 6B 6B 6B 2C 6B
% 25 6C 6C 6C 25 6C
_ 5F 6D 6D 6D 5F 6D
> 3E 6E 6E 6E 3E 6E
? 3F 6F 6F 6F 3F 6F
<o WITH STROKE> F8 70 70 70 C3.B8 8B.67
<E WITH ACUTE> C9 71 71 71 C3.89 8A.4A
<E WITH CIRCUMFLEX> CA 72 72 72 C3.8A 8A.51
<E WITH DIAERESIS> CB 73 73 73 C3.8B 8A.52
<E WITH GRAVE> C8 74 74 74 C3.88 8A.49
<I WITH ACUTE> CD 75 75 75 C3.8D 8A.54
<I WITH CIRCUMFLEX> CE 76 76 76 C3.8E 8A.55
<I WITH DIAERESIS> CF 77 77 77 C3.8F 8A.56
<I WITH GRAVE> CC 78 78 78 C3.8C 8A.53
` 60 79 79 4A 60 79 ##
: 3A 7A 7A 7A 3A 7A
# 23 7B 7B 7B 23 7B
@ 40 7C 7C 7C 40 7C
' 27 7D 7D 7D 27 7D
= 3D 7E 7E 7E 3D 7E
```

```
" 22 7F 7F 7F 22 7F
<O WITH STROKE> D8 80 80 80 C3.98 8A.67
a 61 81 81 81 61 81
b 62 82 82 82 62 82
c 63 83 83 83 63 83
d 64 84 84 84 64 84
e 65 85 85 85 65 85
f 66 86 86 86 66 86
g 67 87 87 87 67 87
h 68 88 88 88 68 88
i 69 89 89 89 69 89
<LEFT POINTING GUILLEMET> AB 8A 8A 8A C2.AB 80.52
<RIGHT POINTING GUILLEMET> BB 8B 8B 8B C2.BB 80.6A
<SMALL LETTER eth> F0 8C 8C 8C C3.B0 8B.57
<y WITH ACUTE> FD 8D 8D 8D C3.BD 8B.71
<SMALL LETTER thorn> FE 8E 8E 8E C3.BE 8B.72
<PLUS-OR-MINUS SIGN> B1 8F 8F 8F C2.B1 80.58
<DEGREE SIGN> B0 90 90 90 C2.B0 80.57
j 6A 91 91 91 6A 91
k 6B 92 92 92 6B 92
l 6C 93 93 93 6C 93
m 6D 94 94 94 6D 94
n 6E 95 95 95 6E 95
o 6F 96 96 96 6F 96
p 70 97 97 97 70 97
q 71 98 98 98 71 98
r 72 99 99 99 72 99
<FEMININE ORDINAL> AA 9A 9A 9A C2.AA 80.51
<MASC. ORDINAL INDICATOR> BA 9B 9B 9B C2.BA 80.69
<SMALL LIGATURE ae> E6 9C 9C 9C C3.A6 8B.47
<CEDILLA> B8 9D 9D 9D C2.B8 80.67
<CAPITAL LIGATURE AE> C6 9E 9E 9E C3.86 8A.47
<CURRENCY SIGN> A4 9F 9F 9F C2.A4 80.45
<MICRO SIGN> B5 A0 A0 A0 C2.B5 80.64
~ 7E A1 A1 FF 7E A1 ##
s 73 A2 A2 A2 73 A2
t 74 A3 A3 A3 74 A3
u 75 A4 A4 A4 75 A4
v 76 A5 A5 A5 76 A5
w 77 A6 A6 A6 77 A6
x 78 A7 A7 A7 78 A7
y 79 A8 A8 A8 79 A8
z 7A A9 A9 A9 7A A9
<INVERTED "!" > A1 AA AA AA C2.A1 80.42
<INVERTED QUESTION MARK> BF AB AB AB C2.BF 80.73
<CAPITAL LETTER ETH> D0 AC AC AC C3.90 8A.57
[ 5B BA AD BB 5B AD ** ##
<CAPITAL LETTER THORN> DE AE AE AE C3.9E 8A.72
<REGISTERED TRADE MARK> AE AF AF AF C2.AE 80.55
<NOT SIGN> AC 5F B0 BA C2.AC 80.53 ** ##
<POUND SIGN> A3 B1 B1 B1 C2.A3 80.44
<YEN SIGN> A5 B2 B2 B2 C2.A5 80.46
<MIDDLE DOT> B7 B3 B3 B3 C2.B7 80.66
<COPYRIGHT SIGN> A9 B4 B4 B4 C2.A9 80.4A
```

```
<SECTION SIGN> A7 B5 B5 B5 C2.A7 80.48
<PARAGRAPH SIGN> B6 B6 B6 B6 C2.B6 80.65
<FRACTION ONE QUARTER> BC B7 B7 B7 C2.BC 80.70
<FRACTION ONE HALF> BD B8 B8 B8 C2.BD 80.71
<FRACTION THREE QUARTERS> BE B9 B9 B9 C2.BE 80.72
<Y WITH ACUTE> DD AD BA AD C3.9D 8A.71 ** ##
<DIAERESIS> A8 BD BB 79 C2.A8 80.49 ** ##
<MACRON> AF BC BC A1 C2.AF 80.56 ##
] 5D BB BD BD 5D BD **
<ACUTE ACCENT> B4 BE BE BE C2.B4 80.63
<MULTIPLICATION SIGN> D7 BF BF BF C3.97 8A.66
{ 7B C0 C0 FB 7B C0 ##
A 41 C1 C1 C1 41 C1
B 42 C2 C2 C2 42 C2
C 43 C3 C3 C3 43 C3
D 44 C4 C4 C4 44 C4
E 45 C5 C5 C5 45 C5
F 46 C6 C6 C6 46 C6
G 47 C7 C7 C7 47 C7
H 48 C8 C8 C8 48 C8
I 49 C9 C9 C9 49 C9
<SOFT HYPHEN> AD CA CA CA C2.AD 80.54
<o WITH CIRCUMFLEX> F4 CB CB CB C3.B4 8B.63
<o WITH DIAERESIS> F6 CC CC CC C3.B6 8B.65
<o WITH GRAVE> F2 CD CD CD C3.B2 8B.59
<o WITH ACUTE> F3 CE CE CE C3.B3 8B.62
<o WITH TILDE> F5 CF CF CF C3.B5 8B.64
} 7D D0 D0 FD 7D D0 ##
J 4A D1 D1 D1 4A D1
K 4B D2 D2 D2 4B D2
L 4C D3 D3 D3 4C D3
M 4D D4 D4 D4 4D D4
N 4E D5 D5 D5 4E D5
O 4F D6 D6 D6 4F D6
P 50 D7 D7 D7 50 D7
Q 51 D8 D8 D8 51 D8
R 52 D9 D9 D9 52 D9
<SUPERSCRIPT ONE> B9 DA DA DA C2.B9 80.68
<u WITH CIRCUMFLEX> FB DB DB DB C3.BB 8B.6A
<u WITH DIAERESIS> FC DC DC DC C3.BC 8B.70
<u WITH GRAVE> F9 DD DD C0 C3.B9 8B.68 ##
<u WITH ACUTE> FA DE DE DE C3.BA 8B.69
<y WITH DIAERESIS> FF DF DF DF C3.BF 8B.73
\ 5C E0 E0 BC 5C E0 ##
<DIVISION SIGN> F7 E1 E1 E1 C3.B7 8B.66
S 53 E2 E2 E2 53 E2
T 54 E3 E3 E3 54 E3
U 55 E4 E4 E4 55 E4
V 56 E5 E5 E5 56 E5
W 57 E6 E6 E6 57 E6
X 58 E7 E7 E7 58 E7
Y 59 E8 E8 E8 59 E8
Z 5A E9 E9 E9 5A E9
<SUPERSCRIPT TWO> B2 EA EA EA C2.B2 80.59
```

```
<O WITH CIRCUMFLEX> D4 EB EB EB C3.94 8A.63
<O WITH DIAERESIS> D6 EC EC EC C3.96 8A.65
<O WITH GRAVE> D2 ED ED ED C3.92 8A.59
<O WITH ACUTE> D3 EE EE EE C3.93 8A.62
<O WITH TILDE> D5 EF EF EF C3.95 8A.64
0 30 F0 F0 F0 30 F0
1 31 F1 F1 F1 31 F1
2 32 F2 F2 F2 32 F2
3 33 F3 F3 F3 33 F3
4 34 F4 F4 F4 34 F4
5 35 F5 F5 F5 35 F5
6 36 F6 F6 F6 36 F6
7 37 F7 F7 F7 37 F7
8 38 F8 F8 F8 38 F8
9 39 F9 F9 F9 39 F9
<SUPERSCRIPT THREE> B3 FA FA FA C2.B3 80.62
<U WITH CIRCUMFLEX> DB FB FB DD C3.9B 8A.6A ##
<U WITH DIAERESIS> DC FC FC FC C3.9C 8A.70
<U WITH GRAVE> D9 FD FD E0 C3.99 8A.68 ##
<U WITH ACUTE> DA FE FE FE C3.9A 8A.69
<APC> 9F FF FF 5F C2.9F FF ##
```

IDENTIFYING CHARACTER CODE SETS

It is possible to determine which character set you are operating under. But first you need to be really really sure you need to do this. Your code will be simpler and probably just as portable if you don't have to test the character set and do different things, depending. There are actually only very few circumstances where it's not easy to write straight-line code portable to all character sets. See ''Unicode and EBCDIC'' in perluniintro(1) for how to portably specify characters.

But there are some cases where you may want to know which character set you are running under. One possible example is doing sorting in inner loops where performance is critical.

To determine if you are running under ASCII or EBCDIC, you can use the return value of `ord()` or `chr()` to test one or more character values. For example:

```
$is_ascii = "A" eq chr(65)
$is_ebcdic = "A" eq chr(193);
$is_ascii = ord("A") == 65;
$is_ebcdic = ord("A") == 193;
```

There's even less need to distinguish between EBCDIC code pages, but to do so try looking at one or more of the characters that differ between them.

```
$is_ascii = ord('[') == 91;
$is_ebcdic_37 = ord('[') == 186;
$is_ebcdic_1047 = ord('[') == 173;
$is_ebcdic_POSIX_BC = ord('[') == 187;
```

However, it would be unwise to write tests such as:

```
$is_ascii = "\r" ne chr(13) # WRONG
$is_ascii = "\n" ne chr(10) # ILL ADVISED
```

Obviously the first of these will fail to distinguish most ASCII platforms from either a CCSID 0037, a 1047, or a POSIX-BC EBCDIC platform since `"\r"eqchr(13)` under all of those coded character sets. But note too that because `"\n"` is `chr(13)` and `"\r"` is `chr(10)` on old Macintosh (which is an ASCII platform) the second `$is_ascii` test will lead to trouble there.

To determine whether or not perl was built under an EBCDIC code page you can use the Config module like so:

```
    use Config;
    $is_ebcdic = $Config{'ebcdic'} eq 'define';
```

CONVERSIONS

utf8::unicode_to_native() and utf8::native_to_unicode()

These functions take an input numeric code point in one encoding and return what its equivalent value is in the other.

See utf8.

tr///

In order to convert a string of characters from one character set to another a simple list of numbers, such as in the right columns in the above table, along with Perl's `tr///` operator is all that is needed. The data in the table are in ASCII/Latin1 order, hence the EBCDIC columns provide easy-to-use ASCII/Latin1 to EBCDIC operations that are also easily reversed.

For example, to convert ASCII/Latin1 to code page 037 take the output of the second numbers column from the output of recipe 2 (modified to add `"\"` characters), and use it in `tr///` like so:

```
$cp_037 =
'\x00\x01\x02\x03\x37\x2D\x2E\x2F\x16\x05\x25\x0B\x0C\x0D\x0E\x0F' .
'\x10\x11\x12\x13\x3C\x3D\x32\x26\x18\x19\x3F\x27\x1C\x1D\x1E\x1F' .
'\x40\x5A\x7F\x7B\x5B\x6C\x50\x7D\x4D\x5D\x5C\x4E\x6B\x60\x4B\x61' .
'\xF0\xF1\xF2\xF3\xF4\xF5\xF6\xF7\xF8\xF9\x7A\x5E\x4C\x7E\x6E\x6F' .
'\x7C\xC1\xC2\xC3\xC4\xC5\xC6\xC7\xC8\xC9\xD1\xD2\xD3\xD4\xD5\xD6' .
'\xD7\xD8\xD9\xE2\xE3\xE4\xE5\xE6\xE7\xE8\xE9\xBA\xE0\xBB\xB0\x6D' .
'\x79\x81\x82\x83\x84\x85\x86\x87\x88\x89\x91\x92\x93\x94\x95\x96' .
'\x97\x98\x99\xA2\xA3\xA4\xA5\xA6\xA7\xA8\xA9\xC0\x4F\xD0\xA1\x07' .
'\x20\x21\x22\x23\x24\x15\x06\x17\x28\x29\x2A\x2B\x2C\x09\x0A\x1B' .
'\x30\x31\x1A\x33\x34\x35\x36\x08\x38\x39\x3A\x3B\x04\x14\x3E\xFF' .
'\x41\xAA\x4A\xB1\x9F\xB2\x6A\xB5\xBD\xB4\x9A\x8A\x5F\xCA\xAF\xBC' .
'\x90\x8F\xEA\xFA\xBE\xA0\xB6\xB3\x9D\xDA\x9B\x8B\xB7\xB8\xB9\xAB' .
'\x64\x65\x62\x66\x63\x67\x9E\x68\x74\x71\x72\x73\x78\x75\x76\x77' .
'\xAC\x69\xED\xEE\xEB\xEF\xEC\xBF\x80\xFD\xFE\xFB\xFC\xAD\xAE\x59' .
'\x44\x45\x42\x46\x43\x47\x9C\x48\x54\x51\x52\x53\x58\x55\x56\x57' .
'\x8C\x49\xCD\xCE\xCB\xCF\xCC\xE1\x70\xDD\xDE\xDB\xDC\x8D\x8E\xDF';

my $ebcdic_string = $ascii_string;
eval '$ebcdic_string =~ tr/\000-\377/' . $cp_037 . '/';
```

To convert from EBCDIC 037 to ASCII just reverse the order of the tr/// arguments like so:

```
my $ascii_string = $ebcdic_string;
eval '$ascii_string =~ tr/' . $cp_037 . '/\000-\377/';
```

Similarly one could take the output of the third numbers column from recipe 2 to obtain a `$cp_1047` table. The fourth numbers column of the output from recipe 2 could provide a `$cp_posix_bc` table suitable for transcoding as well.

If you wanted to see the inverse tables, you would first have to sort on the desired numbers column as in recipes 4, 5 or 6, then take the output of the first numbers column.

iconv

XPG operability often implies the presence of an *iconv* utility available from the shell or from the C library. Consult your system's documentation for information on iconv.

On OS/390 or z/OS see the *iconv(1)* manpage. One way to invoke the `iconv` shell utility from within perl would be to:

```
# OS/390 or z/OS example
$ascii_data = `echo '$ebcdic_data'| iconv -f IBM-1047 -t ISO8859-1`
```

or the inverse map:

```
# OS/390 or z/OS example
$ebcdic_data = `echo '$ascii_data'| iconv -f ISO8859-1 -t IBM-1047`
```

For other Perl-based conversion options see the `Convert::*` modules on CPAN.

C RTL

The OS/390 and z/OS C run-time libraries provide `_atoe()` and `_etoa()` functions.

OPERATOR DIFFERENCES

The `..` range operator treats certain character ranges with care on EBCDIC platforms. For example the following array will have twenty six elements on either an EBCDIC platform or an ASCII platform:

```
@alphabet = ('A'..'Z'); # $#alphabet == 25
```

The bitwise operators such as & ^ | may return different results when operating on string or character data in a Perl program running on an EBCDIC platform than when run on an ASCII platform. Here is an example adapted from the one in perlop:

```
# EBCDIC-based examples
print "j p \n" ^ " a h"; # prints "JAPH\n"
print "JA" | " ph\n"; # prints "japh\n"
print "JAPH\nJunk" & "\277\277\277\277\277"; # prints "japh\n";
print 'p N$' ^ " E<H\n"; # prints "Perl\n";
```

An interesting property of the 32 C0 control characters in the ASCII table is that they can ‘‘literally’’ be constructed as control characters in Perl, e.g. `(chr(0)` eq `\c@)`> `(chr(1)` eq `\cA)`>, and so on. Perl on EBCDIC platforms has been ported to take `\c@` to `chr(0)` and `\cA` to `chr(1)`, etc. as well, but the characters that result depend on which code page you are using. The table below uses the standard acronyms for the controls. The POSIX-BC and 1047 sets are identical throughout this range and differ from the 0037 set at only one spot (21 decimal). Note that the line terminator character may be generated by `\cJ` on ASCII platforms but by `\cU` on 1047 or POSIX-BC platforms and cannot be generated as a `"\c.letter."` control character on 0037 platforms. Note also that `\c\` cannot be the final element in a string or regex, as it will absorb the terminator. But `\c\`*X* is a `FILE SEPARATOR` concatenated with *X* for all *X*. The outlier`\c?` on ASCII, which yields a non-C0 control `DEL`, yields the outlier control `APC` on EBCDIC, the one that isn’t in the block of contiguous controls. Note that a subtlety of this is that `\c?` on ASCII platforms is an ASCII character, while it isn’t equivalent to any ASCII character in EBCDIC platforms.

```
chr ord 8859-1 0037 1047 && POSIX-BC
------------------------------------------------------------------------
\c@ 0 <NUL> <NUL> <NUL>
\cA 1 <SOH> <SOH> <SOH>
\cB 2 <STX> <STX> <STX>
\cC 3 <ETX> <ETX> <ETX>
\cD 4 <EOT> <ST> <ST>
\cE 5 <ENQ> <HT> <HT>
\cF 6 <ACK> <SSA> <SSA>
\cG 7 <BEL> <DEL> <DEL>
\cH 8 <BS> <EPA> <EPA>
\cI 9 <HT> <RI> <RI>
\cJ 10 <LF> <SS2> <SS2>
\cK 11 <VT> <VT> <VT>
\cL 12 <FF> <FF> <FF>
\cM 13 <CR> <CR> <CR>
\cN 14 <SO> <SO> <SO>
\cO 15 <SI> <SI> <SI>
\cP 16 <DLE> <DLE> <DLE>
\cQ 17 <DC1> <DC1> <DC1>
\cR 18 <DC2> <DC2> <DC2>
```

```
\cS 19 <DC3> <DC3> <DC3>
\cT 20 <DC4> <OSC> <OSC>
\cU 21 <NAK> <NEL> <LF> **
\cV 22 <SYN> <BS> <BS>
\cW 23 <ETB> <ESA> <ESA>
\cX 24 <CAN> <CAN> <CAN>
\cY 25 <EOM> <EOM> <EOM>
\cZ 26 <SUB> <PU2> <PU2>
\c[ 27 <ESC> <SS3> <SS3>
\c\X 28 <FS>X <FS>X <FS>X
\c] 29 <GS> <GS> <GS>
\cˆ 30 <RS> <RS> <RS>
\c_ 31 <US> <US> <US>
\c? * <DEL> <APC> <APC>
```

* Note: `\c?` maps to ordinal 127 (`DEL`) on ASCII platforms, but since ordinal 127 is a not a control character on EBCDIC machines, `\c?` instead maps on them to `APC`, which is 255 in 0037 and 1047, and 95 in POSIX-BC.

FUNCTION DIFFERENCES

`chr()` `chr()` must be given an EBCDIC code number argument to yield a desired character return value on an EBCDIC platform. For example:

```
$CAPITAL_LETTER_A = chr(193);
```

`ord()` `ord()` will return EBCDIC code number values on an EBCDIC platform. For example:

```
$the_number_193 = ord("A");
```

`pack()`

The `"c"` and `"C"` templates for `pack()` are dependent upon character set encoding. Examples of usage on EBCDIC include:

```
$foo = pack("CCCC",193,194,195,196);
# $foo eq "ABCD"
$foo = pack("C4",193,194,195,196);
# same thing

$foo = pack("ccxxcc",193,194,195,196);
# $foo eq "AB\0\0CD"
```

The `"U"` template has been ported to mean ‘‘Unicode’’ on all platforms so that

```
pack("U", 65) eq 'A'
```

is true on all platforms. If you want native code points for the low 256, use the `"W"` template. This means that the equivalences

```
pack("W", ord($character)) eq $character
unpack("W", $character) == ord $character
```

will hold.

`print()`

One must be careful with scalars and strings that are passed to print that contain ASCII encodings. One common place for this to occur is in the output of the MIME type header for CGI script writing. For example, many Perl programming guides recommend something similar to:

```
print "Content-type:\ttext/html\015\012\015\012";
# this may be wrong on EBCDIC
```

You can instead write

```
print "Content-type:\ttext/html\r\n\r\n"; # OK for DGW et al
```

and have it work portably.

That is because the translation from EBCDIC to ASCII is done by the web server in this case. Consult your web server's documentation for further details.

`printf()`
: The formats that can convert characters to numbers and vice versa will be different from their ASCII counterparts when executed on an EBCDIC platform. Examples include:

```
printf("%c%c%c",193,194,195); # prints ABC
```

`sort()`
: EBCDIC sort results may differ from ASCII sort results especially for mixed case strings. This is discussed in more detail below.

`sprintf()`
: See the discussion of `"printf()"` above. An example of the use of sprintf would be:

```
$CAPITAL_LETTER_A = sprintf("%c",193);
```

`unpack()`
: See the discussion of `"pack()"` above.

Note that it is possible to write portable code for these by specifying things in Unicode numbers, and using a conversion function:

```
printf("%c",utf8::unicode_to_native(65)); # prints A on all
# platforms
print utf8::native_to_unicode(ord("A")); # Likewise, prints 65
```

See ''Unicode and EBCDIC'' in perluniintro(1) and ''CONVERSIONS'' for other options.

REGULAR EXPRESSION DIFFERENCES

You can write your regular expressions just like someone on an ASCII platform would do. But keep in mind that using octal or hex notation to specify a particular code point will give you the character that the EBCDIC code page natively maps to it. (This is also true of all double-quoted strings.) If you want to write portably, just use the `\N{U+...}` notation everywhere where you would have used `\x{...}`, and don't use octal notation at all.

Starting in Perl v5.22, this applies to ranges in bracketed character classes. If you say, for example, `qr/[\N{U+20}-\N{U+7F}]/`, it means the characters `\N{U+20}`, `\N{U+21}`, ..., `\N{U+7F}`. This range is all the printable characters that the ASCII character set contains.

Prior to v5.22, you couldn't specify any ranges portably, except (starting in Perl v5.5.3) all subsets of the `[A-Z]` and `[a-z]` ranges are specially coded to not pick up gap characters. For example, characters such as ''ô'' (`o WITH CIRCUMFLEX`) that lie between ''I'' and ''J'' would not be matched by the regular expression range `/[H-K]/`. But if either of the range end points is explicitly numeric (and neither is specified by `\N{U+...}`), the gap characters are matched:

```
/[\x89-\x91]/
```

will match `\x8e`, even though `\x89` is ''i'' and `\x91` is ''j'', and `\x8e` is a gap character, from the alphabetic viewpoint.

Another construct to be wary of is the inappropriate use of hex (unless you use `\N{U+...}`) or octal constants in regular expressions. Consider the following set of subs:

```
sub is_c0 {
my $char = substr(shift,0,1);
$char =~ /[\000-\037]/;
}

sub is_print_ascii {
```

```
my $char = substr(shift,0,1);
$char =~ /[\040-\176]/;
}

sub is_delete {
my $char = substr(shift,0,1);
$char eq "\177";
}

sub is_c1 {
my $char = substr(shift,0,1);
$char =~ /[\200-\237]/;
}

sub is_latin_1 { # But not ASCII; not C1
my $char = substr(shift,0,1);
$char =~ /[\240-\377]/;
}
```

These are valid only on ASCII platforms. Starting in Perl v5.22, simply changing the octal constants to equivalent `\N{U+...}` values makes them portable:

```
sub is_c0 {
my $char = substr(shift,0,1);
$char =~ /[\N{U+00}-\N{U+1F}]/;
}

sub is_print_ascii {
my $char = substr(shift,0,1);
$char =~ /[\N{U+20}-\N{U+7E}]/;
}

sub is_delete {
my $char = substr(shift,0,1);
$char eq "\N{U+7F}";
}

sub is_c1 {
my $char = substr(shift,0,1);
$char =~ /[\N{U+80}-\N{U+9F}]/;
}

sub is_latin_1 { # But not ASCII; not C1
my $char = substr(shift,0,1);
$char =~ /[\N{U+A0}-\N{U+FF}]/;
}
```

And here are some alternative portable ways to write them:

```
sub Is_c0 {
my $char = substr(shift,0,1);
return $char =~ /[[:cntrl:]]/a && ! Is_delete($char);

# Alternatively:
# return $char =~ /[[:cntrl:]]/
# && $char =~ /[[:ascii:]]/
# && ! Is_delete($char);
```

```
}

sub Is_print_ascii {
my $char = substr(shift,0,1);

return $char =~ /[[:print:]]/a;

# Alternatively:
# return $char =~ /[[:print:]]/ && $char =~ /[[:ascii:]]/;

# Or
# return $char
# =~ /[ !"\#\$%&'()*+,\-.\/0-9:;<=>?\@A-Z[\\\]^_`a-z{|}~]/;
}

sub Is_delete {
my $char = substr(shift,0,1);
return utf8::native_to_unicode(ord $char) == 0x7F;
}

sub Is_c1 {
use feature 'unicode_strings';
my $char = substr(shift,0,1);
return $char =~ /[[:cntrl:]]/ && $char !~ /[[:ascii:]]/;
}

sub Is_latin_1 { # But not ASCII; not C1
use feature 'unicode_strings';
my $char = substr(shift,0,1);
return ord($char) < 256
&& $char !~ /[[:ascii:]]/
&& $char !~ /[[:cntrl:]]/;
}
```

Another way to write `Is_latin_1()` would be to use the characters in the range explicitly:

```
sub Is_latin_1 {
my $char = substr(shift,0,1);
$char =~ /[¡¢£¤¥¦§¨©ª«¬®¯°±²³´µ¶·¸¹º»¼½¾¿ÀÁÂÃÄÅÆÇÈÉÊËÌÍÎÏ]
[ÐÑÒÓÔÕÖ×ØÙÚÛÜÝÞßàáâãäåæçèéêëìíîïðñòóôõö÷øùúûüýþÿ]/x;
}
```

Although that form may run into trouble in network transit (due to the presence of 8 bit characters) or on non ISO-Latin character sets. But it does allow `Is_c1` to be rewritten so it works on Perls that don't have `'unicode_strings'` (earlier than v5.14):

```
sub Is_latin_1 { # But not ASCII; not C1
my $char = substr(shift,0,1);
return ord($char) < 256
&& $char !~ /[[:ascii:]]/
&& ! Is_latin1($char);
}
```

SOCKETS

Most socket programming assumes ASCII character encodings in network byte order. Exceptions can include CGI script writing under a host web server where the server may take care of translation for you. Most host web servers convert EBCDIC data to ISO-8859-1 or Unicode on output.

SORTING

One big difference between ASCII-based character sets and EBCDIC ones are the relative positions of the characters when sorted in native order. Of most concern are the upper- and lowercase letters, the digits, and the underscore ("_"). On ASCII platforms the native sort order has the digits come before the uppercase letters which come before the underscore which comes before the lowercase letters. On EBCDIC, the underscore comes first, then the lowercase letters, then the uppercase ones, and the digits last. If sorted on an ASCII-based platform, the two-letter abbreviation for a physician comes before the two letter abbreviation for drive; that is:

```
@sorted = sort(qw(Dr. dr.)); # @sorted holds ('Dr.','dr.') on ASCII,
# but ('dr.','Dr.') on EBCDIC
```

The property of lowercase before uppercase letters in EBCDIC is even carried to the Latin 1 EBCDIC pages such as 0037 and 1047. An example would be that ‘‘Ë’’ (`E WITH DIAERESIS`, 203) comes before ‘‘ë’’ (`e WITH DIAERESIS`, 235) on an ASCII platform, but the latter (83) comes before the former (115) on an EBCDIC platform. (Astute readers will note that the uppercase version of ‘‘ß’’ `SMALL LETTER SHARP S` is simply ‘ ‘SS’’ and that the upper case versions of ‘‘ÿ’’ (small `y WITH DIAERESIS`) and ‘‘µ’’ (`MICRO SIGN`) are not in the 0..255 range but are in Unicode, in a Unicode enabled Perl).

The sort order will cause differences between results obtained on ASCII platforms versus EBCDIC platforms. What follows are some suggestions on how to deal with these differences.

Ignore ASCII vs. EBCDIC sort differences.

This is the least computationally expensive strategy. It may require some user education.

Use a sort helper function

This is completely general, but the most computationally expensive strategy. Choose one or the other character set and transform to that for every sort comparision. Here’s a complete example that transforms to ASCII sort order:

```
sub native_to_uni($) {
my $string = shift;

# Saves time on an ASCII platform
return $string if ord 'A' == 65;

my $output = "";
for my $i (0 .. length($string) - 1) {
$output
.= chr(utf8::native_to_unicode(ord(substr($string, $i, 1))));
}

# Preserve utf8ness of input onto the output, even if it didn't need
# to be utf8
utf8::upgrade($output) if utf8::is_utf8($string);

return $output;
}

sub ascii_order { # Sort helper
return native_to_uni($a) cmp native_to_uni($b);
}

sort ascii_order @list;
```

MONO CASE then sort data (for non-digits, non-underscore)

If you don’t care about where digits and underscore sort to, you can do something like this

```
sub case_insensitive_order { # Sort helper
return lc($a) cmp lc($b)
}

sort case_insensitive_order @list;
```

If performance is an issue, and you don't care if the output is in the same case as the input, Use `tr///` to transform to the case most employed within the data. If the data are primarily UPPERCASE non-Latin1, then apply `tr/[a-z]/[A-Z]/`, and then `sort()`. If the data are primarily lowercase non Latin1 then apply `tr/[A-Z]/[a-z]/` before sorting. If the data are primarily UPPERCASE and include Latin-1 characters then apply:

```
tr/[a-z]/[A-Z]/;
tr/[àáâãäåæçèéêëìíîïðñòóôõöùúûüýþ]/[ÀÁÂÃÄÅÆÇÈÉÊËÌÍÎÏÐÑÒÓÔÕÖØÙÚÛÜÝÞ/;
s/ß/SS/g;
```

then `sort()`. If you have a choice, it's better to lowercase things to avoid the problems of the two Latin-1 characters whose uppercase is outside Latin-1: "ÿ" (small `y WITH DIAERESIS`) and "µ" (`MICRO SIGN`). If you do need to upppercase, you can; with a Unicode-enabled Perl, do:

```
tr/ÿ/\x{178}/;
tr/µ/\x{39C}/;
```

Perform sorting on one type of platform only.

This strategy can employ a network connection. As such it would be computationally expensive.

TRANSFORMATION FORMATS

There are a variety of ways of transforming data with an intra character set mapping that serve a variety of purposes. Sorting was discussed in the previous section and a few of the other more popular mapping techniques are discussed next.

URL decoding and encoding

Note that some URLs have hexadecimal ASCII code points in them in an attempt to overcome character or protocol limitation issues. For example the tilde character is not on every keyboard hence a URL of the form:

```
http://www.pvhp.com/~pvhp/
```

may also be expressed as either of:

```
http://www.pvhp.com/%7Epvhp/

http://www.pvhp.com/%7epvhp/
```

where 7E is the hexadecimal ASCII code point for "~". Here is an example of decoding such a URL in any EBCDIC code page:

```
    $url = 'http://www.pvhp.com/%7Epvhp/';
$url =~ s/%([0-9a-fA-F]{2})/
pack("c",utf8::unicode_to_native(hex($1)))/xge;
```

Conversely, here is a partial solution for the task of encoding such a URL in any EBCDIC code page:

```
    $url = 'http://www.pvhp.com/~pvhp/';
# The following regular expression does not address the
# mappings for: ('.' => '%2E', '/' => '%2F', ':' => '%3A')
$url =~ s/([\t "#%&\(\),;<=>\?\@\[\\\]^`{|}~])/
sprintf("%%%02X",utf8::native_to_unicode(ord($1)))/xge;
```

where a more complete solution would split the URL into components and apply a full s/// substitution only to the appropriate parts.

uu encoding and decoding

The `u` template to `pack()` or `unpack()` will render EBCDIC data in EBCDIC characters equivalent to their ASCII counterparts. For example, the following will print ‘‘Yes indeed\n’’ on either an ASCII or EBCDIC computer:

```
$all_byte_chrs = '';
for (0..255) { $all_byte_chrs .= chr($_); }
$uuencode_byte_chrs = pack('u', $all_byte_chrs);
($uu = <<'ENDOFHEREDOC') =~ s/^\s*//gm;
M``$"`P0%!@<("0H+#`T.#Q`1$A,4%187&!D:&QP='A\@(2(C)"4F)R@I*BLL
M+2XO,#$R,S0U-C<X.3H[/#T^/T!!0D-$149'2$E*2TQ-3D]045)35%565UA9
M6EM<75Y?8&%B8V1E9F=H:6IK;&UN;W!Q<G-T=79W>'EZ>WQ]?G^`@8*#A(6&
MAXB)BHN,C8Z/D)&2DY25EI>8F9J;G)V>GZ"AHJ.DI::GJ*FJJZRMKJ^PL;*S
MM+6VM[BYNKN\O;Z_P,'"P\3%QL?(R<K+S,W.S]#1TM/4U=;7V-G:V]S=WM_@
?X>+CY.7FY^CIZNOL[>[O\/'R\_3U]O?X^?K[_/W^_P``
ENDOFHEREDOC
if ($uuencode_byte_chrs eq $uu) {
print "Yes ";
}
$uudecode_byte_chrs = unpack('u', $uuencode_byte_chrs);
if ($uudecode_byte_chrs eq $all_byte_chrs) {
print "indeed\n";
}
```

Here is a very spartan uudecoder that will work on EBCDIC:

```
#!/usr/local/bin/perl
$_ = <> until ($mode,$file) = /^begin\s*(\d*)\s*(\S*)/;
open(OUT, "> $file") if $file ne "";
while(<>) {
last if /^end/;
next if /[a-z]/;
next unless int((((utf8::native_to_unicode(ord()) - 32 ) & 077)
+ 2) / 3)
== int(length() / 4);
print OUT unpack("u", $_);
}
close(OUT);
chmod oct($mode), $file;
```

Quoted-Printable encoding and decoding

On ASCII-encoded platforms it is possible to strip characters outside of the printable set using:

```
# This QP encoder works on ASCII only
$qp_string =~ s/([=\x00-\x1F\x80-\xFF])/
sprintf("=%02X",ord($1))/xge;
```

Starting in Perl v5.22, this is trivially changeable to work portably on both ASCII and EBCDIC platforms.

```
# This QP encoder works on both ASCII and EBCDIC
$qp_string =~ s/([=\N{U+00}-\N{U+1F}\N{U+80}-\N{U+FF}])/
sprintf("=%02X",ord($1))/xge;
```

For earlier Perls, a QP encoder that works on both ASCII and EBCDIC platforms would look somewhat like the following:

```
$delete = utf8::unicode_to_native(ord("\x7F"));
$qp_string =~
s/([^[:print:]$delete])/
sprintf("=%02X",utf8::native_to_unicode(ord($1)))/xage;
```

(although in production code the substitutions might be done in the EBCDIC branch with the function call and separately in the ASCII branch without the expense of the identity map; in Perl v5.22, the identity map is optimized out so there is no expense, but the alternative above is simpler and is also available in v5.22).

Such QP strings can be decoded with:

```
# This QP decoder is limited to ASCII only
$string =~ s/=([[:xdigit:][[:xdigit:])/chr hex $1/ge;
$string =~ s/=[\n\r]+$//;
```

Whereas a QP decoder that works on both ASCII and EBCDIC platforms would look somewhat like the following:

```
$string =~ s/=([[:xdigit:][:xdigit:]])/
chr utf8::native_to_unicode(hex $1)/xge;
$string =~ s/=[\n\r]+$//;
```

Caesarean ciphers

The practice of shifting an alphabet one or more characters for encipherment dates back thousands of years and was explicitly detailed by Gaius Julius Caesar in his **Gallic Wars** text. A single alphabet shift is sometimes referred to as a rotation and the shift amount is given as a number $n after the string 'rot' or ''rot$n''. Rot0 and rot26 would designate identity maps on the 26-letter English version of the Latin alphabet. Rot13 has the interesting property that alternate subsequent invocations are identity maps (thus rot13 is its own non-trivial inverse in the group of 26 alphabet rotations). Hence the following is a rot13 encoder and decoder that will work on ASCII and EBCDIC platforms:

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

while(<>){
tr/n-za-mN-ZA-M/a-zA-Z/;
print;
}
```

In one-liner form:

```
perl -ne 'tr/n-za-mN-ZA-M/a-zA-Z/;print'
```

Hashing order and checksums

Perl deliberately randomizes hash order for security purposes on both ASCII and EBCDIC platforms.

EBCDIC checksums will differ for the same file translated into ASCII and vice versa.

I18N AND L10N

Internationalization (I18N) and localization (L10N) are supported at least in principle even on EBCDIC platforms. The details are system-dependent and discussed under the ''OS ISSUES'' section below.

MULTI-OCTET CHARACTER SETS

Perl works with UTF-EBCDIC, a multi-byte encoding. In Perls earlier than v5.22, there may be various bugs in this regard.

Legacy multi byte EBCDIC code pages XXX.

OS ISSUES

There may be a few system-dependent issues of concern to EBCDIC Perl programmers.

OS/400

PASE The PASE environment is a runtime environment for OS/400 that can run executables built for PowerPC AIX in OS/400; see perlos400. PASE is ASCII-based, not EBCDIC-based as the ILE.

IFS access
: XXX.

OS/390, z/OS
: Perl runs under Unix Systems Services or USS.

`sigaction`
: `SA_SIGINFO` can have segmentation faults.

`chcp` **chcp** is supported as a shell utility for displaying and changing one's code page. See also *chcp(1)*.

dataset access
: For sequential data set access try:

```
my @ds_records = `cat //DSNAME`;
```

or:

```
my @ds_records = `cat //'HLQ.DSNAME'`;
```

See also the OS390::Stdio module on CPAN.

`iconv` **iconv** is supported as both a shell utility and a C RTL routine. See also the *iconv(1)* and *iconv(3)* manual pages.

locales Locales are supported. There may be glitches when a locale is another EBCDIC code page which has some of the code-page variant characters in other positions.

There aren't currently any real UTF-8 locales, even though some locale names contain the string "UTF-8".

See perllocale(1) for information on locales. The L10N files are in */usr/nls/locale*. `$Config{d_setlocale}` is `'define'` on OS/390 or z/OS.

POSIX-BC?
: XXX.

BUGS

- Not all shells will allow multiple `-e` string arguments to perl to be concatenated together properly as recipes in this document 0, 2, 4, 5, and 6 might seem to imply.
- There are a significant number of test failures in the CPAN modules shipped with Perl v5.22 and 5.24. These are only in modules not primarily maintained by Perl 5 porters. Some of these are failures in the tests only: they don't realize that it is proper to get different results on EBCDIC platforms. And some of the failures are real bugs. If you compile and do a `make test` on Perl, all tests on the `/cpan` directory are skipped.

  In particular, the (now deprecated) encoding pragma is not supported under EBCDIC.

  Encode partially works.
- In earlier Perl versions, when byte and character data were concatenated, the new string was sometimes created by decoding the byte strings as *ISO 8859-1 (Latin-1)*, even if the old Unicode string used EBCDIC.

SEE ALSO

perllocale(1), perlfunc(1), perlunicode(1), utf8.

REFERENCES

<http://anubis.dkuug.dk/i18n/charmaps>

<http://www.unicode.org/>

<http://www.unicode.org/unicode/reports/tr16/>

<http://www.wps.com/projects/codes/> **ASCII: American Standard Code for Information Infiltration**

Tom Jennings, September 1999.

The Unicode Standard, Version 3.0 The Unicode Consortium, Lisa Moore ed., ISBN 0-201-61633-5, Addison Wesley Developers Press, February 2000.

CDRA: IBM - Character Data Representation Architecture - Reference and Registry, IBM SC09-2190-00, December 1996.

‘‘Demystifying Character Sets’’, Andrea Vine, Multilingual Computing & Technology, **#26 Vol. 10 Issue 4**, August/September 1999; ISSN 1523-0309; Multilingual Computing Inc. Sandpoint ID, USA.

Codes, Ciphers, and Other Cryptic and Clandestine Communication Fred B. Wrixon, ISBN 1-57912-040-7, Black Dog & Leventhal Publishers, 1998.

<http://www.bobbemer.com/P-BIT.HTM> **IBM - EBCDIC and the P-bit; The biggest Computer Goof Ever** Robert Bemer.

HISTORY

15 April 2001: added UTF-8 and UTF-EBCDIC to main table, pvhp.

AUTHOR

Peter Prymmer pvhp@best.com wrote this in 1999 and 2000 with CCSID 0819 and 0037 help from Chris Leach and André Pirard A.Pirard@ulg.ac.be as well as POSIX-BC help from Thomas Dorner Thomas.Dorner@start.de. Thanks also to Vickie Cooper, Philip Newton, William Raffloer, and Joe Smith. Trademarks, registered trademarks, service marks and registered service marks used in this document are the property of their respective owners.

Now maintained by Perl5 Porters.