

NAME

perlebcdic - Considerations for running Perl on EBCDIC platforms

DESCRIPTION

An exploration of some of the issues facing Perl programmers on EBCDIC based computers. We do not cover localization, internationalization, or multi-byte character set issues other than some discussion of UTF-8 and UTF-EBCDIC.

Portions that are still incomplete are marked with XXX.

Perl used to work on EBCDIC machines, but there are now areas of the code where it doesn't. If you want to use Perl on an EBCDIC machine, please let us know by sending mail to perlbug@perl.org

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 imply character interpretation by the display and other systems of computers. The range 0..127 can be covered by setting the 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.

There are many character sets that extend the range of integers from 0..2⁷-1 up to 2⁸-1, or 8 bit bytes (octets if you prefer). One common one 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 European languages many of which are based on the Roman, or Latin, alphabet.

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 different from ASCII or 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. The layout on the cards was such that high bits were set for the upper and lower case alphabet characters [a-z] and [A-Z], but there were gaps within each Latin alphabet range.

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 some 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. They are:

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 237 places, in other words they agree on only 19 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.

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.

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. This causes a problem with the semantics of the pack/unpack "U", which are supposed to pack Unicode code points to characters and back to numbers. The problem is: which code points to use for code points less than 256? (for 256 and over there's no problem: Unicode code points are used) In EBCDIC, for the low 256 the EBCDIC code points are used. This means that the equivalences

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

will hold. (If Unicode code points were applied consistently over all the possible code points, pack("U",ord("A")) would in EBCDIC equal *A with acute* or chr(101), and unpack("U", "A") would equal 65, or *non-breaking space*, not 193, or ord "A".)

Remaining Perl Unicode problems in EBCDIC

- Many of the remaining problems seem to be related to case-insensitive matching
- The extensions Unicode::Collate and Unicode::Normalized are not supported under EBCDIC, likewise for the encoding pragma.

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.

You may see the term *invariant* character or code point. This simply means that the character has the same numeric value when encoded as when not. (Note that this is a very different concept from

The 13 variant characters mentioned above.) For example, the ordinal value of 'A' is 193 in most EBCDIC code pages, and also is 193 when encoded in UTF-EBCDIC. All variant code points occupy at least two bytes when encoded. In 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 (80..9f on ASCII platforms).)

A string encoded in UTF-EBCDIC may be longer (but never shorter) than one encoded in UTF-8.

Using Encode

Starting from Perl 5.8 you can use the standard new 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 new 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).

<SHIFT OUT>	14	14	14	14
14	14			
<SHIFT IN>	15	15	15	15
15	15			
<DATA LINK ESCAPE>	16	16	16	16
16	16			
<DEVICE CONTROL ONE>	17	17	17	17
17	17			
<DEVICE CONTROL TWO>	18	18	18	18
18	18			
<DEVICE CONTROL THREE>	19	19	19	19
19	19			
<DEVICE CONTROL FOUR>	20	60	60	60
20	60			
<NEGATIVE ACKNOWLEDGE>	21	61	61	61
21	61			
<SYNCHRONOUS IDLE>	22	50	50	50
22	50			
<END OF TRANSMISSION BLOCK>	23	38	38	38
23	38			
<CANCEL>	24	24	24	24
24	24			
<END OF MEDIUM>	25	25	25	25
25	25			
<SUBSTITUTE>	26	63	63	63
26	63			
<ESCAPE>	27	39	39	39
27	39			
<FILE SEPARATOR>	28	28	28	28
28	28			
<GROUP SEPARATOR>	29	29	29	29
29	29			
<RECORD SEPARATOR>	30	30	30	30
30	30			
<UNIT SEPARATOR>	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	###				
<DELETE>				127	7	7	7
	127	7					
<PADDING CHARACTER>				128	32	32	32
	194.128	32					
<HIGH OCTET PRESET>				129	33	33	33
	194.129	33					
<BREAK PERMITTED HERE>				130	34	34	34
	194.130	34					
<NO BREAK HERE>				131	35	35	35
	194.131	35					
<INDEX>				132	36	36	36
	194.132	36					
<NEXT LINE>				133	21	37	37
	194.133	37	***				
<START OF SELECTED AREA>				134	6	6	6
	194.134	6					
<END OF SELECTED AREA>				135	23	23	23
	194.135	23					
<CHARACTER TABULATION SET>				136	40	40	40
	194.136	40					
<CHARACTER TABULATION WITH JUSTIFICATION>				137	41	41	41
	194.137	41					
<LINE TABULATION SET>				138	42	42	42
	194.138	42					
<PARTIAL LINE FORWARD>				139	43	43	43
	194.139	43					
<PARTIAL LINE BACKWARD>				140	44	44	44
	194.140	44					
<REVERSE LINE FEED>				141	9	9	9
	194.141	9					
<SINGLE SHIFT TWO>				142	10	10	10
	194.142	10					
<SINGLE SHIFT THREE>				143	27	27	27
	194.143	27					
<DEVICE CONTROL STRING>				144	48	48	48
	194.144	48					
<PRIVATE USE ONE>				145	49	49	49
	194.145	49					
<PRIVATE USE TWO>				146	26	26	26
	194.146	26					
<SET TRANSMIT STATE>				147	51	51	51
	194.147	51					
<CANCEL CHARACTER>				148	52	52	52
	194.148	52					
<MESSAGE WAITING>				149	53	53	53
	194.149	53					
<START OF GUARDED AREA>				150	54	54	54
	194.150	54					
<END OF GUARDED AREA>				151	8	8	8
	194.151	8					
<START OF STRING>				152	56	56	56
	194.152	56					
<SINGLE GRAPHIC CHARACTER INTRODUCER>				153	57	57	57
	194.153	57					

<SINGLE CHARACTER INTRODUCER> 194.154 58	154	58	58	58
<CONTROL SEQUENCE INTRODUCER> 194.155 59	155	59	59	59
<STRING TERMINATOR> 194.156 4	156	4	4	4
<OPERATING SYSTEM COMMAND> 194.157 20	157	20	20	20
<PRIVACY MESSAGE> 194.158 62	158	62	62	62
<APPLICATION PROGRAM COMMAND> 194.159 255 ###	159	255	255	95
<NON-BREAKING SPACE> 194.160 128.65	160	65	65	65
<INVERTED EXCLAMATION MARK> 194.161 128.66	161	170	170	170
<CENT SIGN> 194.162 128.67 ###	162	74	74	176
<POUND SIGN> 194.163 128.68	163	177	177	177
<CURRENCY SIGN> 194.164 128.69	164	159	159	159
<YEN SIGN> 194.165 128.70	165	178	178	178
<BROKEN BAR> 194.166 128.71 ###	166	106	106	208
<SECTION SIGN> 194.167 128.72	167	181	181	181
<DIAERESIS> 194.168 128.73 *** ###	168	189	187	121
<COPYRIGHT SIGN> 194.169 128.74	169	180	180	180
<FEMININE ORDINAL INDICATOR> 194.170 128.81	170	154	154	154
<LEFT POINTING GUILLEMET> 194.171 128.82	171	138	138	138
<NOT SIGN> 194.172 128.83 *** ###	172	95	176	186
<SOFT HYPHEN> 194.173 128.84	173	202	202	202
<REGISTERED TRADE MARK SIGN> 194.174 128.85	174	175	175	175
<MACRON> 194.175 128.86 ###	175	188	188	161
<DEGREE SIGN> 194.176 128.87	176	144	144	144
<PLUS-OR-MINUS SIGN> 194.177 128.88	177	143	143	143
<SUPERSCRIPT TWO> 194.178 128.89	178	234	234	234
<SUPERSCRIPT THREE> 194.179 128.98	179	250	250	250
<ACUTE ACCENT> 194.180 128.99	180	190	190	190
<MICRO SIGN> 194.181 128.100	181	160	160	160

<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> 195.146 138.89	210	237	237	237
<O WITH ACUTE> 195.147 138.98	211	238	238	238
<O WITH CIRCUMFLEX> 195.148 138.99	212	235	235	235
<O WITH TILDE> 195.149 138.100	213	239	239	239
<O WITH DIAERESIS> 195.150 138.101	214	236	236	236
<MULTIPLICATION SIGN> 195.151 138.102	215	191	191	191
<O WITH STROKE> 195.152 138.103	216	128	128	128
<U WITH GRAVE> 195.153 138.104 ###	217	253	253	224
<U WITH ACUTE> 195.154 138.105	218	254	254	254
<U WITH CIRCUMFLEX> 195.155 138.106 ###	219	251	251	221
<U WITH DIAERESIS> 195.156 138.112	220	252	252	252
<Y WITH ACUTE> 195.157 138.113 *** ###	221	173	186	173
<CAPITAL LETTER THORN> 195.158 138.114	222	174	174	174
<SMALL LETTER SHARP S> 195.159 138.115	223	89	89	89
<a WITH GRAVE> 195.160 139.65	224	68	68	68
<a WITH ACUTE> 195.161 139.66	225	69	69	69
<a WITH CIRCUMFLEX> 195.162 139.67	226	66	66	66
<a WITH TILDE> 195.163 139.68	227	70	70	70
<a WITH DIAERESIS> 195.164 139.69	228	67	67	67
<a WITH RING ABOVE> 195.165 139.70	229	71	71	71
<SMALL LIGATURE ae> 195.166 139.71	230	156	156	156
<c WITH CEDILLA> 195.167 139.72	231	72	72	72
<e WITH GRAVE> 195.168 139.73	232	84	84	84
<e WITH ACUTE> 195.169 139.74	233	81	81	81
<e WITH CIRCUMFLEX> 195.170 139.81	234	82	82	82
<e WITH DIAERESIS> 195.171 139.82	235	83	83	83
<i WITH GRAVE> 195.172 139.83	236	88	88	88
<i WITH ACUTE> 195.173 139.84	237	85	85	85

<i WITH CIRCUMFLEX> 195.174 139.85	238	86	86	86
<i WITH DIAERESIS> 195.175 139.86	239	87	87	87
<SMALL LETTER eth> 195.176 139.87	240	140	140	140
<n WITH TILDE> 195.177 139.88	241	73	73	73
<o WITH GRAVE> 195.178 139.89	242	205	205	205
<o WITH ACUTE> 195.179 139.98	243	206	206	206
<o WITH CIRCUMFLEX> 195.180 139.99	244	203	203	203
<o WITH TILDE> 195.181 139.100	245	207	207	207
<o WITH DIAERESIS> 195.182 139.101	246	204	204	204
<DIVISION SIGN> 195.183 139.102	247	225	225	225
<o WITH STROKE> 195.184 139.103	248	112	112	112
<u WITH GRAVE> 195.185 139.104 ###	249	221	221	192
<u WITH ACUTE> 195.186 139.105	250	222	222	222
<u WITH CIRCUMFLEX> 195.187 139.106	251	219	219	219
<u WITH DIAERESIS> 195.188 139.112	252	220	220	220
<y WITH ACUTE> 195.189 139.113	253	141	141	141
<SMALL LETTER thorn> 195.190 139.114	254	142	142	142
<y WITH DIAERESIS> 195.191 139.115	255	223	223	223

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(/.{43}\d{1,3}\s{6,8}\d{1,3}\s{6,8}\d{1,3}\s{6,8}\d{1,3}/)'\
-e '{push(@l,$_)}' \
-e 'END{print map{$_->[0]}' \
-e '          sort{$a->[1] <=> $b->[1]}' \
-e '          map{[$_,substr($_,52,3)]@l;}' perlebcdic.pod
```

If you would rather see it in CCSID 1047 order then change the number 52 in the last line to 61, like this:

recipe 5

```
perl \
-ne 'if(/.{43}\d{1,3}\s{6,8}\d{1,3}\s{6,8}\d{1,3}\s{6,8}\d{1,3}/)'\
-e '{push(@l,$_)}' \
```

```
-e 'END{print map{$_->[0]}' \
-e '          sort{$a->[1] <=> $b->[1]}' \
-e '          map{[$_ , substr($_, 61, 3)]}@1;}' perlebcdic.pod
```

If you would rather see it in POSIX-BC order then change the number 61 in the last line to 70, like this:

recipe 6

```
perl \
-ne 'if(/.{43}\d{1,3}\s{6,8}\d{1,3}\s{6,8}\d{1,3}\s{6,8}\d{1,3}/)'\
-e '{push(@l,$_)}' \
-e 'END{print map{$_->[0]}' \
-e '          sort{$a->[1] <=> $b->[1]}' \
-e '          map{[$_ , substr($_, 70, 3)]}@1;}' perlebcdic.pod
```

IDENTIFYING CHARACTER CODE SETS

To determine the character set you are running under from perl one could 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);
```

Also, `"\t"` is a HORIZONTAL TABULATION character so that:

```
$is_ascii = ord("\t") == 9;
$is_ebcdic = ord("\t") == 5;
```

To distinguish EBCDIC code pages try looking at one or more of the characters that differ between them. For example:

```
$is_ebcdic_37 = "\n" eq chr(37);
$is_ebcdic_1047 = "\n" eq chr(21);
```

Or better still choose a character that is uniquely encoded in any of the code sets, e.g.:

```
$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" eq chr(13)` under all of those coded character sets. But note too that because `"\n" is chr(13)` and `"\r" is chr(10)` on the 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

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 thirty three 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 FEED` 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`.

chr	ord	8859-1	0037	1047 && POSIX-BC
<code>\c?</code>	127		"	"
<code>\c@</code>	0	<NUL>	<NUL>	<NUL>
<code>\cA</code>	1	<SOH>	<SOH>	<SOH>
<code>\cB</code>	2	<STX>	<STX>	<STX>
<code>\cC</code>	3	<ETX>	<ETX>	<ETX>
<code>\cD</code>	4	<EOT>	<ST>	<ST>
<code>\cE</code>	5	<ENQ>	<HT>	<HT>
<code>\cF</code>	6	<ACK>	<SSA>	<SSA>
<code>\cG</code>	7	<BEL>		
<code>\cH</code>	8	<BS>	<EPA>	<EPA>
<code>\cI</code>	9	<HT>	<RI>	<RI>
<code>\cJ</code>	10	<LF>	<SS2>	<SS2>
<code>\cK</code>	11	<VT>	<VT>	<VT>
<code>\cL</code>	12	<FF>	<FF>	<FF>
<code>\cM</code>	13	<CR>	<CR>	<CR>
<code>\cN</code>	14	<SO>	<SO>	<SO>
<code>\cO</code>	15	<SI>	<SI>	<SI>
<code>\cP</code>	16	<DLE>	<DLE>	<DLE>
<code>\cQ</code>	17	<DC1>	<DC1>	<DC1>
<code>\cR</code>	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>	
\c^	30	<RS>	<RS>	<RS>	
\c_	31	<US>	<US>	<US>	

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"
```

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
```

Under the IBM OS/390 USS Web Server or WebSphere on z/OS for example you should instead write that as:

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

That is because the translation from EBCDIC to ASCII is done by the web server in this case (such code will not be appropriate for the Macintosh however). 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.

REGULAR EXPRESSION DIFFERENCES

As of perl 5.005_03 the letter range regular expressions such as [A-Z] and [a-z] have been especially coded to not pick up gap characters. For example, characters such as `ô` WITH CIRCUMFLEX that lie between I and J would not be matched by the regular expression range `/[H-K]/`. This works in the other direction, too, if either of the range end points is explicitly numeric: `[\x89-\x91]` will match `\x8e`, even though `\x89` is i and `\x91` is j, and `\x8e` is a gap character from the alphabetic viewpoint.

If you do want to match the alphabet gap characters in a single octet regular expression try matching the hex or octal code such as `/\313/` on EBCDIC or `/\364/` on ASCII platforms to have your regular expression match `ô` WITH CIRCUMFLEX.

Another construct to be wary of is the inappropriate use of hex 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 {
    my $char = substr(shift,0,1);
    $char =~ /[\240-\377]/;
```

}

The above would be adequate if the concern was only with numeric code points. However, the concern may be with characters rather than code points and on an EBCDIC platform it may be desirable for constructs such as `if (is_print_ascii("A")) {print "A is a printable character\n";}` to print out the expected message. One way to represent the above collection of character classification subs that is capable of working across the four coded character sets discussed in this document is as follows:

```

sub Is_c0 {
    my $char = substr(shift,0,1);
    if (ord('^')==94) { # ascii
        return $char =~ /\000-\037/;
    }
    if (ord('^')==176) { # 0037
        return $char =~
/[\000-\003\067\055-\057\026\005\045\013-\023\074\075\062\046\030\031\077\0
47\034-\037]/;
    }
    if (ord('^')==95 || ord('^')==106) { # 1047 || posix-bc
        return $char =~
/[\000-\003\067\055-\057\026\005\025\013-\023\074\075\062\046\030\031\077\0
47\034-\037]/;
    }
}

sub Is_print_ascii {
    my $char = substr(shift,0,1);
    $char =~ /[ !"#\$\%&'()*+,-.\0-9:;<=>?@A-Z[\ \ \ ]^_`a-z{|}~]/;
}

sub Is_delete {
    my $char = substr(shift,0,1);
    if (ord('^')==94) { # ascii
        return $char eq "\177";
    }
    else { # ebcdic
        return $char eq "\007";
    }
}

sub Is_c1 {
    my $char = substr(shift,0,1);
    if (ord('^')==94) { # ascii
        return $char =~ /\200-\237/;
    }
    if (ord('^')==176) { # 0037
        return $char =~
/[\040-\044\025\006\027\050-\054\011\012\033\060\061\032\063-\066\010\070-\0
073\040\024\076\377]/;
    }
    if (ord('^')==95) { # 1047
        return $char =~
/[\040-\045\006\027\050-\054\011\012\033\060\061\032\063-\066\010\070-\073\0
40\024\076\377]/;
    }
}

```

```

    }
    if (ord('^')==106) { # posix-bc
        return $char =~

/[\040-\045\006\027\050-\054\011\012\033\060\061\032\063-\066\010\070-\073\
040\024\076\137]//;
    }
}

sub Is_latin_1 {
    my $char = substr(shift,0,1);
    if (ord('^')==94) { # ascii
        return $char =~ /[\240-\377]//;
    }
    if (ord('^')==176) { # 0037
        return $char =~

/[\101\252\112\261\237\262\152\265\275\264\232\212\137\312\257\274\220\217\
352\372\276\240\266\263\235\332\233\213\267\270\271\253\144\145\142\146\143
\147\236\150\164\161-\163\170\165-\167\254\151\355\356\353\357\354\277\200\
375\376\373\374\255\256\131\104\105\102\106\103\107\234\110\124\121-\123\13
0\125-\127\214\111\315\316\313\317\314\341\160\335\336\333\334\215\216\337]
//;
    }
    if (ord('^')==95) { # 1047
        return $char =~

/[\101\252\112\261\237\262\152\265\273\264\232\212\260\312\257\274\220\217\
352\372\276\240\266\263\235\332\233\213\267\270\271\253\144\145\142\146\143
\147\236\150\164\161-\163\170\165-\167\254\151\355\356\353\357\354\277\200\
375\376\373\374\272\256\131\104\105\102\106\103\107\234\110\124\121-\123\13
0\125-\127\214\111\315\316\313\317\314\341\160\335\336\333\334\215\216\337]
//;
    }
    if (ord('^')==106) { # posix-bc
        return $char =~

/[\101\252\260\261\237\262\320\265\171\264\232\212\272\312\257\241\220\217\
352\372\276\240\266\263\235\332\233\213\267\270\271\253\144\145\142\146\143
\147\236\150\164\161-\163\170\165-\167\254\151\355\356\353\357\354\277\200\
340\376\335\374\255\256\131\104\105\102\106\103\107\234\110\124\121-\123\13
0\125-\127\214\111\315\316\313\317\314\341\160\300\336\333\334\215\216\337]
//;
    }
}

```

Note however that only the `Is_ascii_print()` sub is really independent of coded character set. 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 =~
/[\u0000-\u0005\u0007-\u0008\u0009-\u000a\u000b-\u000c\u000d-\u000e\u000f\u0010-\u001f\u0020-\u0021\u0022-\u0023\u0024-\u0025\u0026-\u0027\u0028-\u0029\u002a-\u002b\u002c-\u002d\u002e-\u002f\u0030-\u0039\u0040-\u0041\u0042-\u0043\u0044-\u0045\u0046-\u0047\u0048-\u0049\u004a-\u004b\u004c-\u004d\u004e-\u004f\u0050-\u0051\u0052-\u0053\u0054-\u0055\u0056-\u0057\u0058-\u0059\u005a-\u005b\u005c-\u005d\u005e-\u005f\u0060-\u0061\u0062-\u0063\u0064-\u0065\u0066-\u0067\u0068-\u0069\u006a-\u006b\u006c-\u006d\u006e-\u006f\u0070-\u0071\u0072-\u0073\u0074-\u0075\u0076-\u0077\u0078-\u0079\u007a-\u007b\u007c-\u007d\u007e-\u007f\u0080-\u0081\u0082-\u0083\u0084-\u0085\u0086-\u0087\u0088-\u0089\u008a-\u008b\u008c-\u008d\u008e-\u008f\u0090-\u0091\u0092-\u0093\u0094-\u0095\u0096-\u0097\u0098-\u0099\u009a-\u009b\u009c-\u009d\u009e-\u009f\u00a0-\u00a1\u00a2-\u00a3\u00a4-\u00a5\u00a6-\u00a7\u00a8-\u00a9\u00aa-\u00ab\u00ac-\u00ad\u00ae-\u00af\u00b0-\u00b1\u00b2-\u00b3\u00b4-\u00b5\u00b6-\u00b7\u00b8-\u00b9\u00ba-\u00bb\u00bc-\u00bd\u00be-\u00bf\u00c0-\u00c1\u00c2-\u00c3\u00c4-\u00c5\u00c6-\u00c7\u00c8-\u00c9\u00ca-\u00cb\u00cc-\u00cd\u00ce-\u00cf\u00d0-\u00d1\u00d2-\u00d3\u00d4-\u00d5\u00d6-\u00d7\u00d8-\u00d9\u00da-\u00db\u00dc-\u00dd\u00de-\u00df\u00e0-\u00e1\u00e2-\u00e3\u00e4-\u00e5\u00e6-\u00e7\u00e8-\u00e9\u00ea-\u00eb\u00ec-\u00ed\u00ee-\u00ef\u00f0-\u00f1\u00f2-\u00f3\u00f4-\u00f5\u00f6-\u00f7\u00f8-\u00f9\u00fa-\u00fb\u00fc-\u00fd\u00fe-\u00ff]//;
}

```

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.

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 upper and lower case letters and the letters compared to the digits. 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 version of `ÿ Y WITH DIAERESIS` is not in the 0..255 range but it is at U+x0178 in Unicode, or "`\x{178}`" 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.

MONO CASE then sort data.

In order to minimize the expense of mono casing mixed-case text, try to `tr///` towards the character set case most employed within the data. If the data are primarily UPPERCASE non Latin 1 then apply `tr/[a-z]/[A-Z]/` then `sort()`. If the data are primarily lowercase non Latin 1 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/[À Á Â Ã Ä Å Æ Ç È É Ê Ë Ì Í Î Ï Ñ Ò Ó Ô Õ Ö × Ø Ù Ú Û Ü Ý Þ ß à á â ã ä å æ ç è é ê ë ì í î ï ñ ò ó ô õ ö ÷ ø ù ú û ü ý þ ß / [ÀĀ•Ā, Āf
Ā,,Ā...Ā†Ā‡ĀˆĀ‰ĀŠĀ<ĀĈĀ•ĀŽĀ•Ā•Ā Ā Ā"Ā"Ā•Ā-Ā~Ā™ĀšĀ>ĀœĀ•Āž / ;
s/Āÿ/SS/g;
```

then `sort()`. Do note however that such Latin-1 manipulation does not address the `ÿ Y WITH DIAERESIS` character that will remain at code point 255 on ASCII platforms, but 223 on most EBCDIC platforms where it will sort to a place less than the EBCDIC numerals. With a Unicode-enabled Perl you might try:

```
tr/^?/\x{178}/;
```

The strategy of mono casing data before sorting does not preserve the case of the data and may not be acceptable for that reason.

Convert, sort data, then re convert.

This is the most expensive proposition that does not employ a network connection.

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 under CCSID 1047:

```
$url = 'http://www.pvhp.com/%7Epvhp/';
# this array assumes code page 1047
my @a2e_1047 = (
    0, 1, 2, 3, 55, 45, 46, 47, 22, 5, 21, 11, 12, 13, 14, 15,
    16, 17, 18, 19, 60, 61, 50, 38, 24, 25, 63, 39, 28, 29, 30, 31,
    64, 90,127,123, 91,108, 80,125, 77, 93, 92, 78,107, 96, 75, 97,
    240,241,242,243,244,245,246,247,248,249,122, 94, 76,126,110,111,
    124,193,194,195,196,197,198,199,200,201,209,210,211,212,213,214,
    215,216,217,226,227,228,229,230,231,232,233,173,224,189, 95,109,
    121,129,130,131,132,133,134,135,136,137,145,146,147,148,149,150,
    151,152,153,162,163,164,165,166,167,168,169,192, 79,208,161, 7,
    32, 33, 34, 35, 36, 37, 6, 23, 40, 41, 42, 43, 44, 9, 10, 27,
    48, 49, 26, 51, 52, 53, 54, 8, 56, 57, 58, 59, 4, 20, 62,255,
    65,170, 74,177,159,178,106,181,187,180,154,138,176,202,175,188,
    144,143,234,250,190,160,182,179,157,218,155,139,183,184,185,171,
    100,101, 98,102, 99,103,158,104,116,113,114,115,120,117,118,119,
    172,105,237,238,235,239,236,191,128,253,254,251,252,186,174, 89,
    68, 69, 66, 70, 67, 71,156, 72, 84, 81, 82, 83, 88, 85, 86, 87,
    140, 73,205,206,203,207,204,225,112,221,222,219,220,141,142,223
);
$url =~ s/%([0-9a-fA-F]{2})/pack("c", $a2e_1047[hex($1)])/ge;
```

Conversely, here is a partial solution for the task of encoding such a URL under the 1047 code page:

```
$url = 'http://www.pvhp.com/~pvhp/';
# this array assumes code page 1047
my @e2a_1047 = (
    0, 1, 2, 3,156, 9,134,127,151,141,142, 11, 12, 13, 14, 15,
    16, 17, 18, 19,157, 10, 8,135, 24, 25,146,143, 28, 29, 30, 31,
    128,129,130,131,132,133, 23, 27,136,137,138,139,140, 5, 6, 7,
    144,145, 22,147,148,149,150, 4,152,153,154,155, 20, 21,158, 26,
    32,160,226,228,224,225,227,229,231,241,162, 46, 60, 40, 43,124,
```

```

    38,233,234,235,232,237,238,239,236,223, 33, 36, 42, 41, 59, 94,
    45, 47,194,196,192,193,195,197,199,209,166, 44, 37, 95, 62, 63,
    248,201,202,203,200,205,206,207,204, 96, 58, 35, 64, 39, 61, 34,
    216, 97, 98, 99,100,101,102,103,104,105,171,187,240,253,254,177,
    176,106,107,108,109,110,111,112,113,114,170,186,230,184,198,164,
    181,126,115,116,117,118,119,120,121,122,161,191,208, 91,222,174,
    172,163,165,183,169,167,182,188,189,190,221,168,175, 93,180,215,
    123, 65, 66, 67, 68, 69, 70, 71, 72, 73,173,244,246,242,243,245,
    125, 74, 75, 76, 77, 78, 79, 80, 81, 82,185,251,252,249,250,255,
    92,247, 83, 84, 85, 86, 87, 88, 89, 90,178,212,214,210,211,213,
    48, 49, 50, 51, 52, 53, 54, 55, 56, 57,179,219,220,217,218,159
);
# The following regular expression does not address the
# mappings for: ( '.' => '%2E', '/' => '%2F', ':' => '%3A')
$url =~ s/([\t
"#%&\(\),;<=>\?@\[\]\\^`{|}~])/sprintf("%02X",$e2a_1047[ord($1)])/ge;

```

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

In the remaining examples a `@e2a` or `@a2e` array may be employed but the assignment will not be shown explicitly. For code page 1047 you could use the `@a2e_1047` or `@e2a_1047` arrays just shown.

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 provided that the `@e2a` array is filled in appropriately:

```

#!/usr/local/bin/perl
@e2a = ( # this must be filled in
);
$_ = <> until ($mode,$file) = /^begin\s*(\d*)\s*(\S*)//;
open(OUT, "> $file") if $file ne "";

```

```

while(<>) {
    last if /^end/;
    next if /[a-z]/;
    next unless int((((e2a[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))/ge;

```

Whereas a QP encoder that works on both ASCII and EBCDIC platforms would look somewhat like the following (where the EBCDIC branch @e2a array is omitted for brevity):

```

if (ord('A') == 65) {      # ASCII
    $delete = "\x7F";     # ASCII
    @e2a = (0 .. 255)     # ASCII to ASCII identity map
}
else {                     # EBCDIC
    $delete = "\x07";     # EBCDIC
    @e2a =                # EBCDIC to ASCII map (as shown above)
}
$qp_string =~
    s/([^\
!"#$%&'()*+,-./0-9:;<>?@A-Z[\]\\^_`a-z{|}~$delete])/sprintf(="%02X", $e
2a[ord($1)])/ge;

```

(although in production code the substitutions might be done in the EBCDIC branch with the @e2a array and separately in the ASCII branch without the expense of the identity map).

Such QP strings can be decoded with:

```

# This QP decoder is limited to ASCII only
$string =~ s/([0-9A-Fa-f][0-9A-Fa-f])/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 (where the @a2e array is omitted for brevity):

```

$string =~ s/([0-9A-Fa-f][0-9A-Fa-f])/chr $a2e[hex $1]/ge;
$string =~ s/[\n\r]+$//;

```

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

To the extent that it is possible to write code that depends on hashing order there may be differences between hashes as stored on an ASCII-based platform and hashes stored on an EBCDIC-based platform. XXX

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" in *perlebcdic* section below.

MULTI-OCTET CHARACTER SETS

Perl may work with an internal UTF-EBCDIC encoding form for wide characters on EBCDIC platforms in a manner analogous to the way that it works with the UTF-8 internal encoding form on ASCII based platforms.

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.

chcp

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

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.

OS/390, z/OS 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

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

VM/ESA?

XXX.

POSIX-BC?

XXX.

BUGS

This pod document contains literal Latin 1 characters and may encounter translation difficulties. In particular one popular nroff implementation was known to strip accented characters to their unaccented counterparts while attempting to view this document through the **pod2man** program (for example, you may see a plain `ÿ` rather than one with a diaeresis as in `ÿ`). Another nroff truncated the resultant manpage at the first occurrence of 8 bit characters.

Not all shells will allow multiple `-e` string arguments to perl to be concatenated together properly as recipes 0, 2, 4, 5, and 6 might seem to imply.

SEE ALSO

perllocale, *perlfunc*, *perlunicode*, *utf8*.

REFERENCES

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

<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.