SMASHING MAGAZINE

All About Unicode, UTF8 & Character Sets

By Paul Tero

Published on June 6th, 2012 in with 60 Comments

This is a story that dates back to the earliest days of computers. The story has a plot, well, sort of. It has competition and intrigue, as well as traversing oodles of countries and languages. There is conflict and resolution, and a happyish ending. But the main focus is the characters — **110,116** of them. By the end of the story, they will all find their own unique place in this world.

This story (or *article*, as known on Smashing Magazine) will follow a few of those characters more closely, as they journey from Web server to browser, and back again. Along the way, you'll find out more about the history of characters, character sets, Unicode and UTF-8, and why question marks and odd accented characters sometimes show up in databases and text files.

Warning: this article contains lots of numbers, including a bit of binary – best approached after your morning cup of coffee.

ASCII

Computers only deal in numbers and not letters, so it's important that all computers agree on which numbers represent which letters.

Let's say my computer used the number 1 for A, 2 for B, 3 for C, etc and yours used 0 for A, 1 for B, etc. If I sent you the message HELLO, then the numbers 8, 5, 12, 12, 15 would whiz across the wires. But for you 8 means I, so you would receive and decode it as IFMMP. To communicate effectively, we would need to agree on a standard way of encoding the characters.

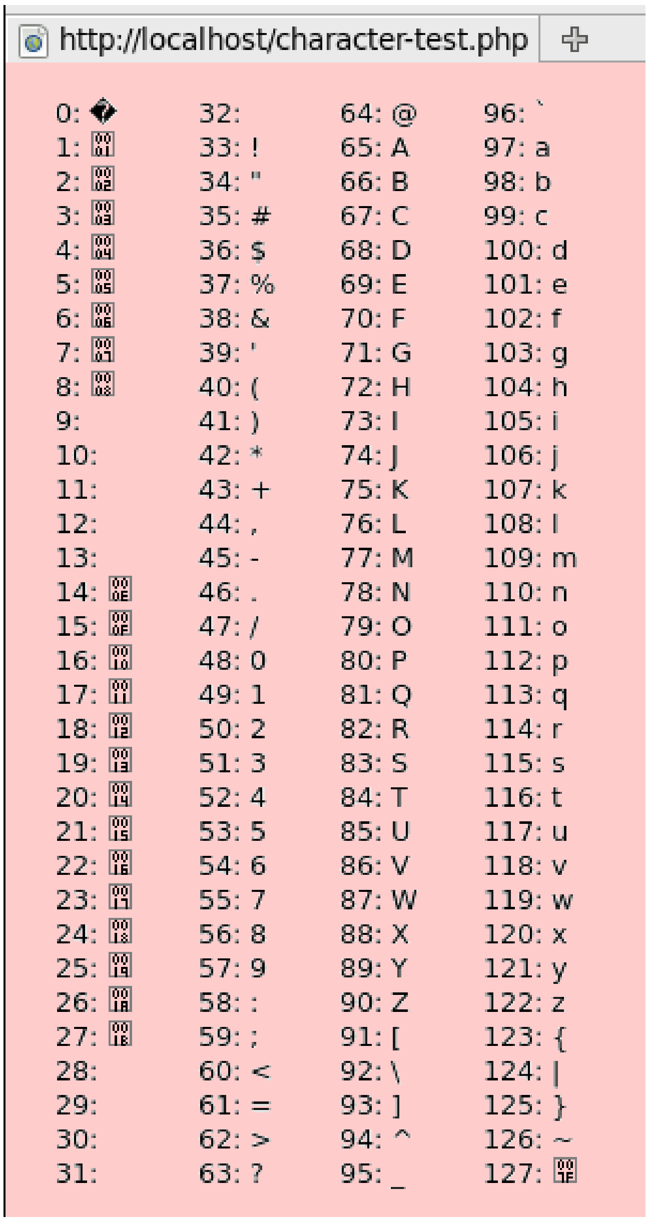
To this end, in the 1960s the American Standards Association created a 7-bit encoding called the American Standard Code for Information Interchange (ASCII[1]). In this encoding HELLO is 72, 69, 76, 76, 79 and would be transmitted digitally as 1001000 1000101 1001100 1001100 1001111. Using 7 bits gives 128 possible values from 0000000 to 1111111, so ASCII has enough room for all lower case and upper case Latin letters, along with each numerical digit, common punctuation marks, spaces, tabs and other control characters. In 1968, US President Lyndon Johnson made it official[2] – all computers must use and understand ASCII.

TRYING IT YOURSELF

There are plenty of ASCII tables[3] available, displaying or describing the 128 characters. Or you can make one of your own with a little bit of CSS, HTML and Javascript, most of which is to get it to display nicely:

```
<html>
<body>
<style type="text/css">p {float: left; padding: 0 15px; margin: 0; font-size: 80%;}</style
>
<script type="text/javascript">
for (var i=0; i<128; i++) document.writeln ((i%32?'':'<p>') + i + ': ' + String.fromCharCo
de (i) + '<br>');
</script>
</body>
</html>
```

This will display a table like this:



http://localhost/character-test.php

```
0: �      32:      64: @    96: `
1: ▯      33: !    65: A    97: a
2: ▯      34: "    66: B    98: b
3: ▯      35: #    67: C    99: c
4: ▯      36: $    68: D    100: d
5: ▯      37: %    69: E    101: e
6: ▯      38: &    70: F    102: f
7: ▯      39: '    71: G    103: g
8: ▯      40: (    72: H    104: h
9:        41: )    73: I    105: i
10:       42: *    74: J    106: j
11:       43: +    75: K    107: k
12:       44: ,    76: L    108: l
13:       45: -    77: M    109: m
14: ▯     46: .    78: N    110: n
15: ▯     47: /    79: O    111: o
16: ▯     48: 0    80: P    112: p
17: ▯     49: 1    81: Q    113: q
18: ▯     50: 2    82: R    114: r
19: ▯     51: 3    83: S    115: s
20: ▯     52: 4    84: T    116: t
21: ▯     53: 5    85: U    117: u
22: ▯     54: 6    86: V    118: v
23: ▯     55: 7    87: W    119: w
24: ▯     56: 8    88: X    120: x
25: ▯     57: 9    89: Y    121: y
26: ▯     58: :    90: Z    122: z
27: ▯     59: ;    91: [    123: {
28:       60: <    92: \    124: |
29:       61: =    93: ]    125: }
30:       62: >    94: ^    126: ~
31:       63: ?    95: _    127: ▯
```

Do-It-Yourself Javascript ASCII table viewed in Firefox

The most important bit of this is the Javascript `String.fromCharCode` function. It takes a number and turns it into a character. In fact, the following four lines of HTML and Javascript all produce the same result. They all get the browser to display character numbers 72, 69, 76, 76 and 79:

```
HELLO
HELLO
<script>document.write ("HELLO");</script>
<script>document.write (String.fromCharCode (72,69,76,76,79));</script>
```

Also notice how Firefox displays the unprintable characters (like backspace and escape) in the first column. Some browsers show blanks or question marks. Firefox squeezes four hexadecimal digits into a small box.

The Eighth Bit

Teleprinters[4] and stock tickers were quite happy sending 7 bits of information to each other. But the new fangled microprocessors[5] of the 1970s preferred to work with powers of 2. They could process 8 bits at a time and so used 8 bits (aka a byte or octet) to store each character, giving 256 possible values.

An 8 bit character can store a number up to 255, but ASCII only assigns up to 127. The other values from 128 to 255 are spare. Initially, IBM PCs used the spare slots to represent accented letters, various symbols and shapes and a handful of Greek letters. For instance, number 200 was the lower left corner of a box: ╚, and 224 was the Greek letter alpha in lower case: **α**. This way of encoding the letters was later given the name code page 437[6].

However, unlike ASCII, characters 128-255 were never standardized, and various countries started using the spare slots for their own alphabets. Not everybody agreed that 224 should display **α**, not even the Greeks. This led to the creation of a handful of new code pages[7]. For example, in Russian IBM computers using code page 885, 224 represents the Cyrillic letter **Я**. And in Greek code page 737, it is lower case omega: **ω**.

Even then there was disagreement. From the 1980s Microsoft Windows introduced its own code pages. In the Cyrillic code page Windows-1251, 224 represents the Cyrillic letter **а**, and **Я** is at 223.

In the late 1990s, an attempt at standardization was made. Fifteen different 8 bit character sets were created to cover many different alphabets such as Cyrillic, Arabic, Hebrew, Turkish, and Thai. They are called ISO-8859-1 up to ISO-8859-16[8] (number 12 was abandoned). In the Cyrillic ISO-8859-5, 224 represents the letter **р**, and **Я** is at 207.

So if a Russian friend sends you a document, you really need to know what code page it uses. The document by itself is just a sequence of numbers. Character 224 could be **Я**, **а** or **р**. Viewed using the wrong code page, it will look like a bunch of scrambled letters and symbols.

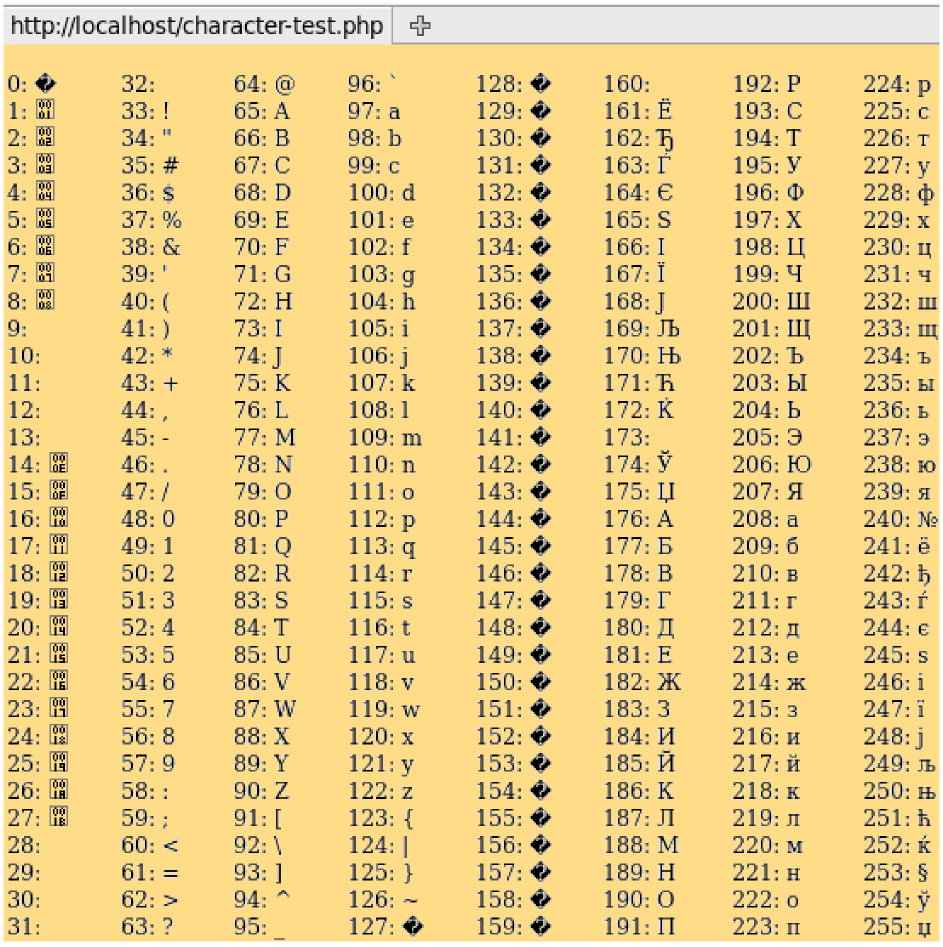
(The situation isn't quite as bad when viewing Web pages – as Web browsers can usually detect a page's character set based on frequency analysis and other such techniques. But this is a false sense of security – they can and do get it wrong.)

TRYING IT YOURSELF

Code pages are also known as **character sets**. You can explore these character sets yourself, but you have to use PHP or a similar server side language this time (roughly because the character needs to be in the page before it gets to the browser). Save these lines in a PHP file and upload it to your server:

```
<html>
<head>
<meta charset="ISO-8859-5">
</head>
<body>
<style type="text/css">p {float: left; padding: 0 15px; margin: 0; font-size: 80%;}</style
>
<?php  for ($i=0; $i<256; $i++) echo ($i%32?'':'<p>') . $i . ': ' . chr ($i) . '<br>'; ?>
</body>
</html>
```

This will display a table like this:



http://localhost/character-test.php

0: �	32:	64: @	96: `	128: �	160:	192: Р	224: р
1: □	33: !	65: A	97: a	129: �	161: Ё	193: С	225: с
2: □	34: "	66: B	98: b	130: �	162: Ђ	194: Т	226: т
3: □	35: #	67: C	99: c	131: �	163: Ѓ	195: У	227: у
4: □	36: $	68: D	100: d	132: �	164: Є	196: Ф	228: ф
5: □	37: %	69: E	101: e	133: �	165: Ѕ	197: Х	229: х
6: □	38: &	70: F	102: f	134: �	166: І	198: Ц	230: ц
7: □	39: '	71: G	103: g	135: �	167: Ї	199: Ч	231: ч
8: □	40: (	72: H	104: h	136: �	168: Ј	200: Ш	232: ш
9:	41:)	73: I	105: i	137: �	169: Љ	201: Щ	233: щ
10:	42: *	74: J	106: j	138: �	170: Њ	202: Ъ	234: ъ
11:	43: +	75: K	107: k	139: �	171: Ћ	203: Ы	235: ы
12:	44: ,	76: L	108: l	140: �	172: Ќ	204: Ь	236: ь
13:	45: -	77: M	109: m	141: �	173:	205: Э	237: э
14: □	46: .	78: N	110: n	142: �	174: Ў	206: Ю	238: ю
15: □	47: /	79: O	111: o	143: �	175: Џ	207: Я	239: я
16: □	48: 0	80: P	112: p	144: �	176: А	208: а	240: №
17: □	49: 1	81: Q	113: q	145: �	177: Б	209: б	241: ё
18: □	50: 2	82: R	114: r	146: �	178: В	210: в	242: ђ
19: □	51: 3	83: S	115: s	147: �	179: Г	211: г	243: ѓ
20: □	52: 4	84: T	116: t	148: �	180: Д	212: д	244: є
21: □	53: 5	85: U	117: u	149: �	181: Е	213: е	245: ѕ
22: □	54: 6	86: V	118: v	150: �	182: Ж	214: ж	246: і
23: □	55: 7	87: W	119: w	151: �	183: З	215: з	247: ї
24: □	56: 8	88: X	120: x	152: �	184: И	216: и	248: ј
25: □	57: 9	89: Y	121: y	153: �	185: Й	217: й	249: љ
26: □	58: :	90: Z	122: z	154: �	186: К	218: к	250: њ
27: □	59: ;	91: [	123: {	155: �	187: Л	219: л	251: ћ
28:	60: <	92: \	124: |	156: �	188: М	220: м	252: ќ
29:	61: =	93:]	125: }	157: �	189: Н	221: н	253: §
30:	62: >	94: ^	126: ~	158: �	190: О	222: о	254: ў
31:	63: ?	95: _	127: �	159: �	191: П	223: п	255: џ

Cyrillic character set ISO-8859-5 viewed in Firefox

The PHP function `chr` does a similar thing to Javascript's `String.fromCharCode`. For example `chr(224)` embeds the number 224 into the Web page before sending it to the browser. As we've seen above, 224 can mean many different things. So, the browser needs to know which character set to use to display the 224. That's what the first line above is for. It tells the browser to use the Cyrillic character set ISO-8858-5:

```
<meta charset="ISO-8859-5">
```

If you exclude the `charset` line, then it will display using the browser's default. In countries with Latin-based alphabets (like the UK and US), this is probably ISO-8859-1, in which case 224 is an a with grave accent: **à**. Try changing this line to ISO-8859-7 or Windows-1251 and refresh the page. You can also override the character set in the browser. In Firefox go to View > Character Encoding. Swap between a few to see what effect it has. If you try to display more than 256 characters, the sequence will repeat.

SUMMARY CIRCA 1990

This is the situation in about 1990. Documents can be written, saved and exchanged in many languages, but **you need to know which character set they use**. There is also no easy way to use two or more non-English alphabets in the same document, and alphabets with more than 256 characters like Chinese and Japanese have to use entirely different systems.

Finally, the Internet is coming! Internationalization and globalization is about to make this a much bigger issue. A new standard is required.

Unicode To The Rescue

Starting in the late 1980s, a new standard was proposed – one that would assign a unique number (officially known as a code point) to every letter in every language, one that would have way more than 256 slots. It was called Unicode[9]. It is now in version 6.1 and consists of over 110,000 code points. If you have a few hours to spare you can watch them all whiz past[10].

The first 128 Unicode code points are the same as ASCII. The range 128-255 contains currency symbols and other common signs and accented characters (aka characters with diacritical marks[11]), and much of it is borrowed ISO-8859-1. After 256 there are many more accented characters. After 880 it gets into Greek letters, then Cyrillic, Hebrew, Arabic, Indic scripts, and Thai. Chinese, Japanese and Korean start from 11904 with many others in between.

This is great – no more ambiguity – each letter is represented by its own unique number. Cyrillic **Я** is always 1071 and Greek **α** is always 945. 224 is always **à**, and **H** is still 72. Note that these Unicode code points are officially written in hexadecimal preceded by U+. So the Unicode code point H is usually written as U+0048 rather than 72 (to convert from hexadecimal to decimal: 4*16+8=72).

The major problem is that there are more than 256 of them. The characters will no longer fit into 8 bits. However **Unicode is not a character set or code page**. So officially that is not the Unicode Consortium's problem. They just came up with the idea and left someone else to sort out the implementation. That will be discussed in the next two sections.

Unicode Inside The Browser

Unicode does not fit into 8 bits, not even into 16. Although only 110,116 code points are in use, it has the capability to define up to 1,114,112 of them, which would require 21 bits.

However, computers have advanced since the 1970s. An 8 bit microprocessor is a bit out of date. New computers now have 64 bit processors, so why can't we move beyond an 8 bit character and into a 32 bit or 64 bit character?

The first answer is: we can!

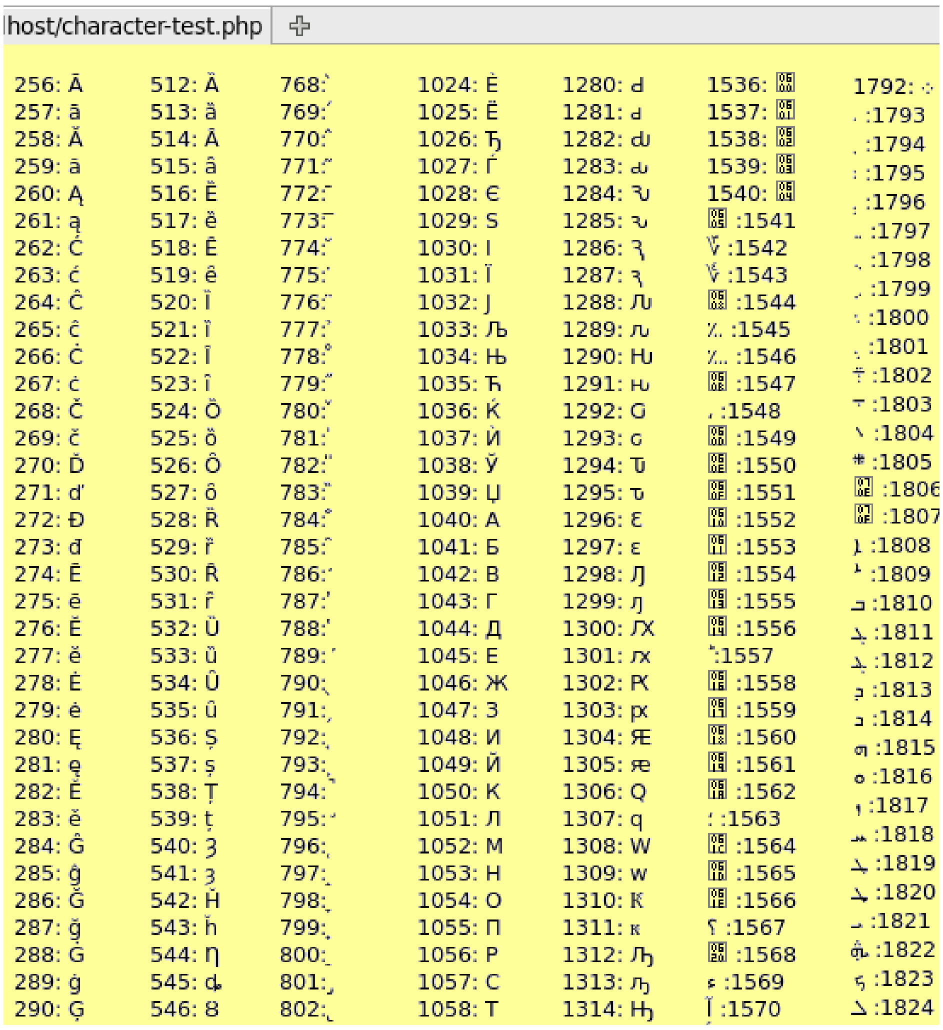
A lot of software is written in C or C++, which supports a "wide character". This is a 32 bit character called `wchar_t`. It is an extension of C's 8 bit `char` type. Internally, modern Web browsers use these wide characters (or something similar) and can theoretically quite happily deal with over 4 billion distinct characters. This is plenty for Unicode. So – **internally, modern Web browers use Unicode**.

TRYING IT YOURSELF

The Javascript code below is similar to the ASCII code above, except it goes up to a much higher number. For each number, it tells the browser to display the corresponding Unicode code point:

```
<html>
<body>
<style type="text/css">p {float: left; padding: 0 15px; margin: 0; font-size: 80%;}</style
>
<script type="text/javascript">
for (var i=0; i<2096; i++)
  document.writeln ((i%256?'':'<p>') + i + ': ' + String.fromCharCode (i) + '<br>');
</script>
</body>
</html>
```

It will output a table like this:


host/character-test.php

```
256: Ā   512: Ȁ   768:̀    1024: Ѐ   1280: Ԁ   1536: ؀    1792: ܀
257: ā   513: ȁ   769:́    1025: Ё   1281: ԁ   1537: ؁    ܁ :1793
258: Ă   514: Ȃ   770:̂    1026: Ђ   1282: Ԃ   1538: ؂    ܂ :1794
259: ă   515: ȃ   771:̃    1027: Ѓ   1283: ԃ   1539: ؃    ܃ :1795
260: Ą   516: Ȅ   772:̄    1028: Є   1284: Ԅ   1540: ؄    ܄ :1796
261: ą   517: ȅ   773:̅    1029: Ѕ   1285: ԅ   ؅ :1541    ܅ :1797
262: Ć   518: Ȇ   774:̆    1030: І   1286: Ԇ   ؆ :1542    ܆ :1798
263: ć   519: ȇ   775:̇    1031: Ї   1287: ԇ   ؇ :1543    ܇ :1799
264: Ĉ   520: Ȉ   776:̈    1032: Ј   1288: Ԉ   ؈ :1544    ܈ :1800
265: ĉ   521: ȉ   777:̉    1033: Љ   1289: ԉ   ؉ :1545    ܉ :1801
266: Ċ   522: Ȋ   778:̊    1034: Њ   1290: Ԋ   ؊ :1546    ܊ :1802
267: ċ   523: ȋ   779:̋    1035: Ћ   1291: ԋ   ؋ :1547    ܋ :1803
268: Č   524: Ȍ   780:̌    1036: Ќ   1292: Ԍ   ، :1548    ܌ :1804
269: č   525: ȍ   781:̍    1037: Ѝ   1293: ԍ   ؍ :1549    ܍ :1805
270: Ď   526: Ȏ   782:̎    1038: Ў   1294: Ԏ   ؎ :1550    ܎ :1806
271: ď   527: ȏ   783:̏    1039: Џ   1295: ԏ   ؏ :1551    ܏ :1807
272: Đ   528: Ȑ   784:̐    1040: А   1296: Ԑ   ؐ :1552    ܐ :1808
273: đ   529: ȑ   785:̑    1041: Б   1297: ԑ   ؑ :1553    ܑ :1809
274: Ē   530: Ȓ   786:̒    1042: В   1298: Ԓ   ؒ :1554    ܒ :1810
275: ē   531: ȓ   787:̓    1043: Г   1299: ԓ   ؓ :1555    ܓ :1811
276: Ĕ   532: Ȕ   788:̔    1044: Д   1300: Ԕ   ؔ :1556    ܔ :1812
277: ĕ   533: ȕ   789:̕    1045: Е   1301: ԕ   ؕ:1557     ܕ :1813
278: Ė   534: Ȗ   790:̖    1046: Ж   1302: Ԗ   ؖ :1558    ܖ :1814
279: ė   535: ȗ   791:̗    1047: З   1303: ԗ   ؗ :1559    ܗ :1815
280: Ę   536: Ș   792:̘    1048: И   1304: Ԙ   ؘ :1560    ܘ :1816
281: ę   537: ș   793:̙    1049: Й   1305: ԙ   ؙ :1561    ܙ :1817
282: Ě   538: Ț   794:̚    1050: К   1306: Ԛ   ؚ :1562    ܚ :1818
283: ě   539: ț   795:̛    1051: Л   1307: ԛ   ؛ :1563    ܛ :1819
284: Ĝ   540: Ȝ   796:̜    1052: М   1308: Ԝ   ؜ :1564    ܜ :1820
285: ĝ   541: ȝ   797:̝    1053: Н   1309: ԝ   ؝ :1565    ܝ :1821
286: Ğ   542: Ȟ   798:̞    1054: О   1310: Ԟ   ؞ :1566    ܞ :1822
287: ğ   543: ȟ   799:̟    1055: П   1311: ԟ   ؟ :1567    ܟ :1823
288: Ġ   544: Ƞ   800:̠    1056: Р   1312: Ԡ   ؠ :1568    ܠ :1824
289: ġ   545: ȡ   801:̡    1057: С   1313: ԡ   ء :1569
290: Ģ   546: Ȣ   802:̢    1058: Т   1314: Ԣ   آ :1570
```

A selection of Unicode code points viewed in Firefox

The screenshot above only shows a subset of the first few thousand code points output by the Javascript. The selection includes some Cyrillic and Arabic characters, displayed right-to-left.

The important point here is that Javascript runs completely in the Web browser where 32 bit characters are perfectly acceptable. The Javascript function `String.fromCharCode(1071)` outputs the Unicode code point 1071 which is the letter **Я**.

Similarly if you put the HTML entity `Я` into an HTML page, a modern Web browser would display **Я**. Numerical HTML entities also refer to Unicode.

On the other hand, the PHP function `chr(1071)` would output a forward slash **/** because the `chr` function only deals with 8 bit numbers up to 256 and repeats itself after that, and 1071%256=47 which has been a / since the 1960s.

UTF-8 To The Rescue

So if browsers can deal with Unicode in 32 bit characters, where is the problem? **The problem is in the sending and receiving, and reading and writing of characters.**

The problem remains because:

1. A lot of existing software and protocols send/receive and read/write 8 bit characters
2. Using 32 bits to send/store English text would quadruple the amount of bandwidth/space required

Although browsers can deal with Unicode internally, you still have to get the data from the Web server to the Web browser and back again, and you need to save it in a file or database somewhere. So you still need a way to make 110,000 Unicode code points fit into just 8 bits.

There have been several attempts to solve this problem such as UCS2 and UTF-16. But the winner in recent years is UTF-8, which stands for Universal Character Set Transformation Format 8 bit.

UTF-8 is a clever. It works a bit like the Shift key on your keyboard. Normally when you press the H on your keyboard a lower case “h” appears on the screen. But if you press Shift first, a capital H will appear.

UTF-8 treats numbers 0-127 as ASCII, 192-247 as Shift keys, and 128-192 as the key to be shifted. For instance, characters 208 and 209 shift you into the Cyrillic range. 208 followed by 175[12] is character 1071, the Cyrillic **Я**. The exact calculation[13] is (208%32)*64 + (175%64) = 1071. Characters 224-239 are like a double shift. 226 followed by 190 and then 128[14] is character 12160: ⾀. 240 and over is a triple shift.

UTF-8 is therefore a multi-byte variable-width encoding. Multi-byte because a single character like **Я** takes more than one byte to specify it. Variable-width because some characters like **H** take only 1 byte and some up to 4.

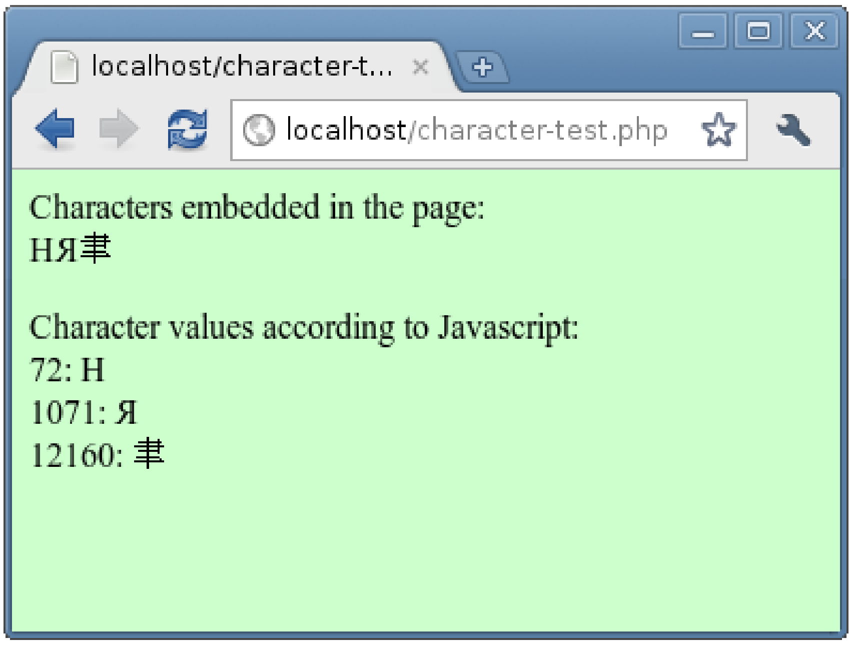
Best of all it is backward compatible with ASCII. Unlike some of the other proposed solutions, any document written only in ASCII, using only characters 0-127, is perfectly valid UTF-8 as well – which saves bandwidth and hassle.

TRYING IT YOURSELF

This is a different experiment. PHP embeds the 6 numbers mentioned above into an HTML page: 72, 208, 175, 226, 190, 128. The browser interprets those numbers as UTF-8, and internally converts them into Unicode code points. Then Javascript outputs the Unicode values. Try changing the character set from UTF-8 to ISO-8859-1 and see what happens:

```
<html>
<head>
<meta charset="UTF-8">
</head>
<body>
<p>Characters embedded in the page:<br>
<span id="chars"><?php echo chr(72).chr(208).chr(175).chr(226).chr(190).chr(128); ?></span
>
<p>Character values according to Javascript:<br>
<script type="text/javascript">
function ShowCharacters (s) {var r=''; for (var i=0; i<s.length; i++)
  r += s.charCodeAt (i) + ': ' + s.substr (i, 1) + '<br>'; return r;}
document.writeln (ShowCharacters (document.getElementById('chars').innerHTML));
</script>
</body>
</html>
```

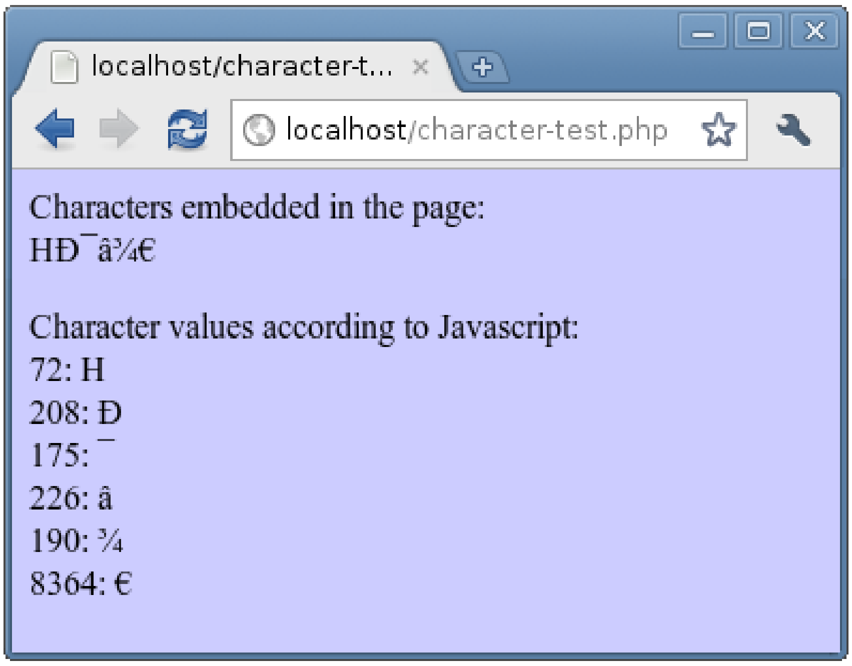
If you are in a hurry, this is what it will look like:



Characters embedded in the page:
HЯ⾀

Character values according to Javascript:
72: H
1071: Я
12160: ⾀

The sequence of numbers above shown using the UTF-8 character set



```
Characters embedded in the page:
HÐ¯â¾€

Character values according to Javascript:
72: H
208: Ð
175: ¯
226: â
190: ¾
8364: €
```

Same sequence of numbers shown using the ISO-8859-1 character set

If you display the page using the UTF-8 character set, you will see only 3 characters: HЯ. If you display it using the character set ISO-8859-1, you will see six separate characters: HÐ¯â¾€ . This is what is happening:

1. On your Web server, PHP is embedding the numbers 72, 208, 175, 226, 190 and 128 into a Web page
2. The Web page whizzes across the Internet from the Web server to your Web browser
3. The browser receives those numbers and interprets them according to the character set
4. The browser internally represents the characters using their Unicode values
5. Javascript outputs the corresponding Unicode values

Notice that when viewed as ISO-8859-1 the first 5 numbers are the same (72, 208, 175, 226, 190) as their Unicode code points. This is because Unicode borrowed heavily from ISO-8859-1 in that range. The last number however, the euro symbol €, is different. It is at position 128 in ISO-8859-1 and has the Unicode value 8364.

SUMMARY CIRCA 2003

UTF-8 is becoming the most popular international character set on the Internet, superseding the older single-byte character sets like ISO-8859-5. When you view or send a non-English document, you still need to know what character set it uses. For widest interoperability, website administrators need to make sure all their web pages use the UTF-8 character sets.

Perhaps the Ð looks familiar – it will sometimes show up if you try to view Russian UTF-8 documents. The next section describes how character sets get confused and end up storing things wrongly in a database.

Lots Of Problems

As long as everybody is speaking UTF-8, this should all work swimmingly. If they aren't, then characters can get mangled. To explain way, imagine a typical interaction a website, such as a user making a comment on a blog post:

1. A Web page displays a comment form
2. The user types a comment and submits.
3. The comment is sent back to the server and saved in a database.
4. The comment is later retrieved from the database and displayed on a Web page

This simple process can go wrong in lots of ways and produce the following types of problems:

HTML ENTITIES

Pretend for a moment that you don't know anything about character sets – erase the last 30 minutes from your memory. The form on your blog will probably display itself using the character set ISO-8859-1. This character set doesn't know any Russian or Thai or Chinese, and only a little bit of Greek. If you attempt to copy and paste any into the form and press Submit, a modern browser will try to convert it into HTML numerical entities like я for **Я**.

That's what will get saved in your database, and that's what will be output when the comment is displayed – which means it will display fine on a Web page, but cause problems when you try to output it to a PDF or email, or run text searches for it in a database.

CONFUSED CHARACTERS

How about if you operate a Russian website, and you have not specified a character set in your Web page? Imagine a Russian user whose default character set is ISO-8859-5. To say "hi", they might type Привет. When the user presses Submit, **the characters are encoded according to the character set of the sending page**. In this case, Привет is encoded as the numbers 191, 224, 216, 210, 213 and 226. Those numbers will get sent across the Internet to the server, and saved like that into a database.

If somebody later views that comment using ISO-8859-5, they will see the correct text. But if they view using a different Russian character set like Windows-1251, they will see їаШТХв. It's still Russian, but makes no sense.

ACCENTED CHARACTERS WITH LOTS OF VOWELS

If someone views the same comment using ISO-8859-1, they will see ¿àØÒÕâ instead of Привет. A longer phrase like Я тоже рада Вас видеть ("nice to see you" in a formal way to a female), submitted as ISO-8859-5, will show up in ISO-8859-1 as Ï âÞÖÕ àÐÔÐ. It looks like that because the 128-255 range of ISO-8859-1 contains lots of vowels with accents.

So if you see this sort of pattern, it's probably because text has been entered in a single byte character set (one of the ISO-8859s or Windows ones) and is being displayed as ISO-8859-1. To fix the text, you'll need to figure out which character set it was entered as, and resubmit it as UTF-8 instead.

ALTERNATING ACCENTED CHARACTERS

What if the user submitted the comment in UTF-8? In that case the Cyrillic characters which make up the word Привет would each get sent as 2 numbers each: 208/159, 209/128, 208/184, 208/178, 208/181 and 209/130. If you viewed that in ISO-8859-1 it would look like: ĐŸÑ€Đ¸Đ²ĐµÑ‚.

Notice that every other character is a Đ or Ñ. Those characters are numbers 208 and 209, and they tell UTF-8 to switch to the Cyrillic range. So if you see a lot of Đ and Ñ, you can assume that you are looking at Russian text entered in UTF-8, viewed as ISO-8859-1. Similarly, Greek will have lots of Î and Ï, 206 and 207. And Hebrew has alternating ×, number 215.

VOWELS BEFORE A POUND AND COPYRIGHT SIGN

A very common issue in the UK is the currency symbol £ getting converted into Â£. This is exactly the same issue as above with a coincidence thrown in to add confusion. The £ symbol has the Unicode and ISO-8859-1 value of 163. Recall that in UTF-8 any character over 127 is represented by a sequence of two or more numbers. In this case, the UTF-8 sequence is 194/163. Mathematically, this is because (194%32)*64 + (**163**%64) = **163**.

Visually it means that the if you view the UTF-8 sequence using ISO-8859-1, it appears to gain a Â which is character 194 in ISO-8859-1. The same thing happens for all Unicode code points 161-191, which includes © and ® and ¥.

So if your £ or © suddenly inherit a Â, it is because they were entered as UTF-8.

BLACK DIAMOND QUESTION MARKS

How about the other way around? If you enter Привет as ISO-8859-5, it will get saved as the numbers shown above: 191, 224, etc. If you then try to view this as UTF-8, you may well see lots of question marks inside black diamonds: �. The browser displays these when it can't make sense of the numbers it is reading.

UTF-8 is self-synchronzising. Unlike other multi-byte character encodings, you always know where you are with UTF-8. If you see a number 192-247, you know you are at the beginning of a multi-byte sequence. If you see 128-191 you know you are in the middle of one. There's no danger of missing the first number and garbling the rest of the text.

This means that in UTF-8, the sequence 191 followed by 224 will never occur naturally, so the browser doesn't know what to do with it and displays �� instead.

This can also cause £ and © related problems. £50 in ISO-8859-1 is the numbers 163, 53 and 48. The 53 and 48 cause no issues, but in UTF-8, 163 can never occur by itself, so this will show up as �50. Similarly if you see �2012, it is probably because ©2012 was input as ISO-8859-1 but is being displayed as UTF-8.

BLANKS, QUESTION MARKS AND BOXES

Even if they are fully up-to-speed with UTF-8 and Unicode, a browser still may not know how to display a character. The first few ASCII characters 1-31 are mostly control sequences for teleprinters (things like Acknowledge and Stop). If you try to display them, a browser might show a ? or a blank or a box with tiny numbers inside it.

Also, Unicode defines over 110,000 characters. **Your browser may not have the correct font to display all of them.** Some of the more obscure characters may also get shown as ? or blank or a small box. In older browsers, even fairly common non-English characters may show as boxes.

Older browsers may also behave differently for some of the issues above, showing ? and blank boxes more often.

DATABASES

The discussion above has avoided the middle step in the process – saving data to a database. Databases like MySQL can also specify a character set for a database, table or column. But it is less important that the Web pages' character set.

When saving and retrieving data, MySQL deals just with numbers. If you tell it to save number 163, it will. If you give it 208/159 it will save those two numbers. And when you retrieve the data, you'll get the same two numbers back.

The character set becomes more important when you use database functions to compare, convert and measure the data. For example, the `LENGTH` of a field may depend on its character set, as do string comparisons using `LIKE` and `=`. The method used to compare strings is called a collation[15].

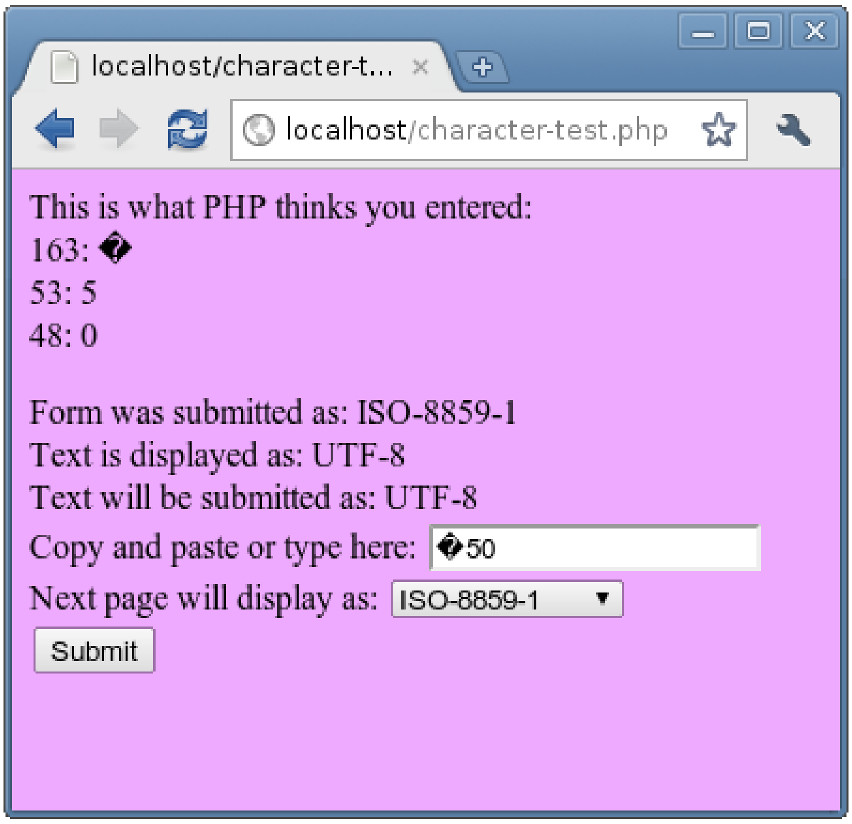
Character sets and collations in MySQL are an in-depth subject. It's not simply a case of changing the character set of a table to UTF-8. There are further SQL commands to take into account to make sure the data goes in and out in the right format as well. This blog[16] is a good starting point.

TRYING IT YOURSELF

The following PHP and Javascript code allows you to experiment with all these issues. You can specify which character set is used to input and output text, and you can see what the browser thinks about it too.

```
<?php
$charset = $_POST['charset']; if (!$charset) $charset = 'ISO-8859-1';
$string = $_POST['string'];
if ($string) {
        echo '<p>This is what PHP thinks you entered:<br>';
        for ($i=0; $i<strlen($string); $i++) {$c=substr ($string,$i,1); echo ord ($c).': '
.$c.' <br/>';}
}
?>
<html>
<head>
<meta charset="<?=$charset?>">
</head>
<body>
<form method="post">
<input name="lastcharset" type="hidden" value="<?php echo $charset?>"/>
Form was submitted as: <?php echo $_POST['lastcharset']?><br/>
Text is displayed as: <?php echo $charset?><br/>
Text will be submitted as: <?php echo $charset?><br/>
Copy and paste or type here:
<input name="string" type="text" size="20" value="<?php echo $string?>"/><br/>
Next page will display as:
<select name="charset"><option>ISO-8859-1<option>ISO-8859-5
<option>Windows-1251<option>ISO-8859-7<option>UTF-8</select><br/>
<input type="submit" value="Submit" onclick="ShowCharacters (this.form.string.value); retu
rn 1;"/>
</form>
<script type="text/javascript">
function ShowCharacters (s) {
  var r='You entered:';
  for (var i=0; i<s.length; i++) r += 'n' + s.charCodeAt (i) + ': ' + s.substr (i, 1);
  alert (r);
}
</script>
</body>
</html>
```

This is an example of the code in action. The numbers at the top are the numerical values of each of the characters and their representation (when viewed individually) in the current character set:



This is what PHP thinks you entered:
163: �
53: 5
48: 0

Form was submitted as: ISO-8859-1
Text is displayed as: UTF-8
Text will be submitted as: UTF-8
Copy and paste or type here: �50
Next page will display as: ISO-8859-1
Submit

Example of inputting and output in different character sets. This shows a £ sign turning into a � in Google Chrome.

The page above shows the previous, current and future character sets. You can use this code to quickly see how text can get really mangled. For example, if you pressed Submit again above, the � has Unicode code point 65533 which is 239/191/189 in UTF-8 and will be displayed as ï¿½50 in ISO-8859-1. So if you ever get £ symbols turning into ï¿½, that is probably the route they took.

Note that the select box at the bottom will change back to ISO-8859-1 each time.

One Solution

All the encoding problems above are caused by text being submitted in one character set and viewed in another. The solution is to **make sure that every page on your website uses UTF-8**. You can do this with one of these lines immediately after the `<head>` tag:

```
<meta charset="UTF-8">
<meta http-equiv="Content-type" content="text/html; charset=UTF-8">
```

It has to be one of the first things in your Web page, as it will cause the browser to look again at the page in a whole new light. For speed and efficiency, it should do this as soon as possible.

You can also specify UTF-8 in your MySQL tables, though to fully use this feature, you'll need to delve deeper.

Note that users can still override the character set in their browsers. This is rare, but does mean that this solution is not guaranteed to work. For extra safety, you could implement a back-end check to ensure data is arriving in the correct format.

EXISTING WEBSITES

If your website has already been collecting text in a variety of languages, then you will also need to convert your existing data into UTF-8. If there is not much of it, you can use a PHP page like the one above to figure out the original character set, and use the browser to convert the data into UTF-8.

If you have lots of data in various character sets, you'll need to first detect the character set and then convert it. In PHP you can use `mb_detect_encoding`[17] to detect and `iconv`[18] to convert. Reading the comments for `mb_detect_encoding`, it looks like quite a fussy function, so be sure to experiment to make sure you are using it properly and getting the right results.

A potentially misleading function is `utf8_decode`[19]. It turns UTF-8 into ISO-8859-1. Any characters not available in ISO-8859-1 (like Cyrillic, Greek, Thai, etc) are turned into question marks. It's misleading because you might have expected more from it, but it does the best it can.

Summary

This article has relied heavily on numbers and has tried to leave no stone unturned. Hopefully it has provided an exhaustive understanding of character sets, Unicode, UTF-8 and the various problems that can arise. The morals of the story are:

- You need to know the character set in order to make sense of non-Latin text
- Internally, browsers use Unicode to represent characters
- Make sure all your Web pages specify the UTF-8 character set

For a slightly different approach to this subject, this 2003 character set article[20] is excellent. Thank you for sticking with this epic journey.

FOOTNOTES

1 http://en.wikipedia.org/wiki/ASCII

2 http://www.presidency.ucsb.edu/ws/index.php?pid=28724#axzz1s2VIkMsP

3 http://www.asciitable.com/

4 http://en.wikipedia.org/wiki/Teleprinter

5 http://en.wikipedia.org/wiki/Microprocessor#8-bit_designs

6 http://en.wikipedia.org/wiki/Code_page_437

7 http://en.wikipedia.org/wiki/Code_pages#IBM_PC_.28OEM.29_code_pages

8 http://en.wikipedia.org/wiki/Iso-8859

9 http://en.wikipedia.org/wiki/Unicode

10 http://www.babelstone.co.uk/Unicode/unicode.html

11 http://en.wikipedia.org/wiki/Diacritic

12 http://en.wikipedia.org/wiki/%D0%AF#Computing_codes

13 http://en.wikipedia.org/wiki/UTF-8#Examples

14 http://www.fileformat.info/info/unicode/char/2F80/index.htm

15 http://dev.mysql.com/doc/refman/5.5/en/charset-general.html

16 http://blog.tremend.ro/2006/09/26/mysql-php-and-utf8/

17 http://www.php.net/manual/en/function.mb-detect-encoding.php

18 http://www.php.net/manual/en/function.iconv.php

19 http://php.net/manual/en/function.utf8-decode.php

20 http://www.joelonsoftware.com/articles/Unicode.html

Paul Tero

Paul Tero is an experienced PHP programmer and server administrator. He developed the Stockashop ecommerce system in 2005 for Sensable Media. He now works part-time maintaining and developing Stockashop, and the rest of the time freelancing from a corner of his living room, and sleeping, eating, having fun, etc. He has also written numerous other open sourcish scripts and programs.

```
Subject: UTF-8 history
From: "Rob 'Commander' Pike" <r (at) google.com>
Date: Wed, 30 Apr 2003 22:32:32 -0700 (Thu 06:32 BST)
To: mkuhn (at) acm.org, henry (at) spsystems.net
Cc: ken (at) entrisphere.com

Looking around at some UTF-8 background, I see the same incorrect
story being repeated over and over.  The incorrect version is:
        1. IBM designed UTF-8.
        2. Plan 9 implemented it.
That's not true.  UTF-8 was designed, in front of my eyes, on a
placemat in a New Jersey diner one night in September or so 1992.

What happened was this.  We had used the original UTF from ISO 10646
to make Plan 9 support 16-bit characters, but we hated it.  We were
close to shipping the system when, late one afternoon, I received a
call from some folks, I think at IBM - I remember them being in Austin
- who were in an X/Open committee meeting.  They wanted Ken and me to
vet their FSS/UTF design.  We understood why they were introducing a
new design, and Ken and I suddenly realized there was an opportunity
to use our experience to design a really good standard and get the
X/Open guys to push it out.  We suggested this and the deal was, if we
could do it fast, OK.  So we went to dinner, Ken figured out the
bit-packing, and when we came back to the lab after dinner we called
the X/Open guys and explained our scheme.  We mailed them an outline
of our spec, and they replied saying that it was better than theirs (I
don't believe I ever actually saw their proposal; I know I don't
remember it) and how fast could we implement it?  I think this was a
Wednesday night and we promised a complete running system by Monday,
which I think was when their big vote was.

So that night Ken wrote packing and unpacking code and I started
tearing into the C and graphics libraries.  The next day all the code
was done and we started converting the text files on the system
itself.  By Friday some time Plan 9 was running, and only running,
what would be called UTF-8.  We called X/Open and the rest, as they
say, is slightly rewritten history.

Why didn't we just use their FSS/UTF?  As I remember, it was because
in that first phone call I sang out a list of desiderata for any such
encoding, and FSS/UTF was lacking at least one - the ability to
synchronize a byte stream picked up mid-run, with less that one
character being consumed before synchronization.  Becuase that was
lacking, we felt free - and were given freedom - to roll our own.

I think the "IBM designed it, Plan 9 implemented it" story originates
in RFC2279.  At the time, we were so happy UTF-8 was catching on we
didn't say anything about the bungled history.  Neither of us is at
the Labs any more, but I bet there's an e-mail thread in the archive
there that would support our story and I might be able to get someone
to dig it out.

So, full kudos to the X/Open and IBM folks for making the opportunity
happen and for pushing it forward, but Ken designed it with me
cheering him on, whatever the history books say.

-rob

Date: Sat, 07 Jun 2003 18:44:05 -0700
From: "Rob `Commander' Pike" <r (at) google.com>
To: Markus Kuhn <Markus.Kuhn (at) cl.cam.ac.uk>
cc: henry (at) spsystems.net, ken (at) entrisphere.com,
   Greger Leijonhufvud <greger (at) friherr.com>
Subject: Re: UTF-8 history
```

I asked Russ Cox to dig through the archives. I have attached his message.
I think you'll agree it supports the story I sent earlier. The mail we
sent to X/Open (I believe Ken did the editing and mailing of that document)
includes a new desideratum #6 about discovering character boundaries.
We'll never know how much the original X/Open proposal influenced us;
the two proposals are very different but do share some characteristics.
I don't remember looking at it in detail, but it was a long time ago.
I very clearly remember Ken writing on the placemat and wished we had
kept it!

-rob

From: Russ Cox <rsc (at) plan9.bell-labs.com>
To: r (at) google.com
Subject: utf digging
Date-Sent: Saturday, June 07, 2003 7:46 PM -0400

bootes's /sys/src/libc/port/rune.c changed from the
division-heavy old utf on sep 4 1992.
the version that made it into the dump
is dated 19:51:55. it was commented
the next day but otherwise remained unchanged
until nov 14 1996, when runelen was sped up by
inspecting the rune explicitly rather than
using runetochar's return value. may 26 2001
was the next and last change, to add runenlen.

here are some mails from your mail boxes
that turn up by grepping for utf. the first
refers to utf.c, which is a copy of a wctomb and mbtowc
that handle the full 6-byte utf-8 encoding of 32-bit runes.
it's quite ugly, with all the logic in control flow.
i assume it became the code in the proposal
as a result of that first mail.

in /usr/ken/utf/xutf i found a copy of what
appears to be the original not-self-synchronizing
encoding proposal, with the utf-8 scheme tacked
onto the end (starting at "We define 7 byte types").
that's also below. the version below is the first,
dated sep 2 23:44:10. it went through a number of
edits to become the second mail below by the
morning of Sep 8.

the mail log shows that second mail going out
as well as taking a while to come back to ken.

```
helix: Sep  8 03:22:13: ken: upas/sendmail: remote inet!xopen.co.uk!xojig
>From ken Tue Sep  8 03:22:07 EDT 1992 (xojig@xopen.co.uk) 6833
helix: Sep  8 03:22:13: ken: upas/sendmail: delivered rob From ken Tue Sep
8 03:22:07 EDT 1992 6833
helix: Sep  8 03:22:16: ken: upas/sendmail: remote pyxis!andrew From ken
Tue Sep  8 03:22:07 EDT 1992 (andrew) 6833
helix: Sep  8 03:22:19: ken: upas/sendmail: remote coma!dmr From ken Tue
Sep  8 03:22:07 EDT 1992 (dmr) 6833
helix: Sep  8 03:25:52: ken: upas/sendmail: delivered rob From ken Tue Sep
8 03:24:58 EDT 1992 141
helix: Sep  8 03:36:13: ken: upas/sendmail: delivered ken From ken Tue Sep
8 03:36:12 EDT 1992 6833
```

enjoy.

```
>From ken Fri Sep  4 03:37:39 EDT 1992
you might want to look at
        /usr/ken/utf/utf.c
and see if you can make it prettier.

>From ken Tue Sep  8 03:22:07 EDT 1992
Here is our modified FSS-UTF proposal.  The words are the same as on
the previous proposal.  My apologies to the author.  The code has been
tested to some degree and should be pretty good shape.  We have
converted Plan 9 to use this encoding and are about to issue a
distribution to an initial set of university users.

File System Safe Universal Character Set Transformation Format (FSS-UTF)
--------------------------------------------------------------------------

With the approval of ISO/IEC 10646 (Unicode) as an international
standard and the anticipated wide spread use of this universal coded
character set (UCS), it is necessary for historically ASCII based
operating systems to devise ways to cope with representation and
handling of the large number of characters that are possible to be
encoded by this new standard.

There are several challenges presented by UCS which must be dealt with
by historical operating systems and the C-language programming
environment.  The most significant of these challenges is the encoding
scheme used by UCS. More precisely, the challenge is the marrying of
the UCS standard with existing programming languages and existing
operating systems and utilities.

The challenges of the programming languages and the UCS standard are
being dealt with by other activities in the industry.  However, we are
still faced with the handling of UCS by historical operating systems
and utilities.  Prominent among the operating system UCS handling
concerns is the representation of the data within the file system.  An
underlying assumption is that there is an absolute requirement to
maintain the existing operating system software investment while at
the same time taking advantage of the use the large number of
characters provided by the UCS.

UCS provides the capability to encode multi-lingual text within a
single coded character set.  However, UCS and its UTF variant do not
protect null bytes and/or the ASCII slash ("/") making these character
encodings incompatible with existing Unix implementations.  The
following proposal provides a Unix compatible transformation format of
UCS such that Unix systems can support multi-lingual text in a single
encoding.  This transformation format encoding is intended to be used
as a file code.  This transformation format encoding of UCS is
intended as an intermediate step towards full UCS support.  However,
since nearly all Unix implementations face the same obstacles in
supporting UCS, this proposal is intended to provide a common and
compatible encoding during this transition stage.


Goal/Objective
--------------

With the assumption that most, if not all, of the issues surrounding
the handling and storing of UCS in historical operating system file
systems are understood, the objective is to define a UCS
transformation format which also meets the requirement of being usable
on a historical operating system file system in a non-disruptive
manner.  The intent is that UCS will be the process code for the
transformation format, which is usable as a file code.
```

```
Criteria for the Transformation Format
--------------------------------------

Below are the guidelines that were used in defining the UCS
transformation format:

        1) Compatibility with historical file systems:

        Historical file systems disallow the null byte and the ASCII
        slash character as a part of the file name.

        2) Compatibility with existing programs:

        The existing model for multibyte processing is that ASCII does
        not occur anywhere in a multibyte encoding.  There should be
        no ASCII code values for any part of a transformation format
        representation of a character that was not in the ASCII
        character set in the UCS representation of the character.

        3) Ease of conversion from/to UCS.

        4) The first byte should indicate the number of bytes to
        follow in a multibyte sequence.

        5) The transformation format should not be extravagant in
        terms of number of bytes used for encoding.

        6) It should be possible to find the start of a character
        efficiently starting from an arbitrary location in a byte
        stream.


Proposed FSS-UTF
----------------

The proposed UCS transformation format encodes UCS values in the range
[0,0x7fffffff] using multibyte characters of lengths 1, 2, 3, 4, 5,
and 6 bytes.  For all encodings of more than one byte, the initial
byte determines the number of bytes used and the high-order bit in
each byte is set.  Every byte that does not start 10xxxxxx is the
start of a UCS character sequence.

An easy way to remember this transformation format is to note that the
number of high-order 1's in the first byte signifies the number of
bytes in the multibyte character:

   Bits  Hex Min  Hex Max  Byte Sequence in Binary
1    7  00000000 0000007f 0vvvvvvv
2   11  00000080 000007FF 110vvvvv 10vvvvvv
3   16  00000800 0000FFFF 1110vvvv 10vvvvvv 10vvvvvv
4   21  00010000 001FFFFF 11110vvv 10vvvvvv 10vvvvvv 10vvvvvv
5   26  00200000 03FFFFFF 111110vv 10vvvvvv 10vvvvvv 10vvvvvv 10vvvvvv
6   31  04000000 7FFFFFFF 1111110v 10vvvvvv 10vvvvvv 10vvvvvv 10vvvvvv
10vvvvvv

The UCS value is just the concatenation of the v bits in the multibyte
encoding.  When there are multiple ways to encode a value, for example
UCS 0, only the shortest encoding is legal.

Below are sample implementations of the C standard wctomb() and
mbtowc() functions which demonstrate the algorithms for converting
from UCS to the transformation format and converting from the
transformation format to UCS. The sample implementations include error
checks, some of which may not be necessary for conformance:
```

```
typedef
struct
{
        int     cmask;
        int     cval;
        int     shift;
        long    lmask;
        long    lval;
} Tab;

static
Tab     tab[] =
{
        0x80,   0x00,   0*6,    0x7F,           0,              /* 1 byte sequence */
        0xE0,   0xC0,   1*6,    0x7FF,          0x80,           /* 2 byte sequence */
        0xF0,   0xE0,   2*6,    0xFFFF,         0x800,          /* 3 byte sequence */
        0xF8,   0xF0,   3*6,    0x1FFFFF,       0x10000,        /* 4 byte sequence */
        0xFC,   0xF8,   4*6,    0x3FFFFFF,      0x200000,       /* 5 byte sequence */
        0xFE,   0xFC,   5*6,    0x7FFFFFFF,     0x4000000,      /* 6 byte sequence */
        0,                                                      /* end of table */
};

int
mbtowc(wchar_t *p, char *s, size_t n)
{
        long l;
        int c0, c, nc;
        Tab *t;

        if(s == 0)
                return 0;

        nc = 0;
        if(n <= nc)
                return -1;
        c0 = *s & 0xff;
        l = c0;
        for(t=tab; t->cmask; t++) {
                nc++;
                if((c0 & t->cmask) == t->cval) {
                        l &= t->lmask;
                        if(l < t->lval)
                                return -1;
                        *p = l;
                        return nc;
                }
                if(n <= nc)
                        return -1;
                s++;
                c = (*s ^ 0x80) & 0xFF;
                if(c & 0xC0)
                        return -1;
                l = (l<<6) | c;
        }
        return -1;
}

int
wctomb(char *s, wchar_t wc)
{
        long l;
        int c, nc;
        Tab *t;

        if(s == 0)
```

```
                return 0;

        l = wc;
        nc = 0;
        for(t=tab; t->cmask; t++) {
                nc++;
                if(l <= t->lmask) {
                        c = t->shift;
                        *s = t->cval | (l>>c);
                        while(c > 0) {
                                c -= 6;
                                s++;
                                *s = 0x80 | ((l>>c) & 0x3F);
                        }
                        return nc;
                }
        }
        return -1;
}
```

```
>From ken Tue Sep  8 03:24:58 EDT 1992
i mailed it out, but it went into a black hole.
i didnt get my copy. it must be hung up on the
internat address with coma down or something.

>From ken Tue Sep  8 03:42:43 EDT 1992
i finally got my copy.
```

--- /usr/ken/utf/xutf from dump of Sep 2 1992 ---

File System Safe Universal Character Set Transformation Format (FSS-UTF)

With the approval of ISO/IEC 10646 (Unicode) as an international standard and the anticipated wide spread use of this universal coded character set (UCS), it is necessary for historically ASCII based operating systems to devise ways to cope with representation and handling of the large number of characters that are possible to be encoded by this new standard.

There are several challenges presented by UCS which must be dealt with by historical operating systems and the C-language programming environment. The most significant of these challenges is the encoding scheme used by UCS. More precisely, the challenge is the marrying of the UCS standard with existing programming languages and existing operating systems and utilities.

The challenges of the programming languages and the UCS standard are being dealt with by other activities in the industry. However, we are still faced with the handling of UCS by historical operating systems and utilities. Prominent among the operating system UCS handling concerns is the representation of the data within the file system. An underlying assumption is that there is an absolute requirement to maintain the existing operating system software investment while at the same time taking advantage of the use the large number of characters provided by the UCS.

UCS provides the capability to encode multi-lingual text within a single coded character set. However, UCS and its UTF variant do not protect null bytes and/or the ASCII slash ("/") making these character encodings incompatible with existing Unix implementations. The following proposal provides a Unix compatible transformation format of UCS such that Unix systems can support multi-lingual text in a single encoding. This

```
   transformation format encoding is intended to be used as a file code.
   This transformation format encoding of UCS is intended as an
   intermediate step towards full UCS support.  However, since nearly all
   Unix implementations face the same obstacles in supporting UCS, this
   proposal is intended to provide a common and compatible encoding during
   this transition stage.


   Goal/Objective
   --------------

   With the assumption that most, if not all, of the issues surrounding the
   handling and storing of UCS in historical operating system file systems
   are understood, the objective is to define a UCS transformation format
   which also meets the requirement of being usable on a historical
   operating system file system in a non-disruptive manner. The intent is
   that UCS will be the process code for the transformation format, which
   is usable as a file code.

   Criteria for the Transformation Format
   --------------------------------------

   Below are the guidelines that were used in defining the UCS
   transformation format:

       1) Compatibility with historical file systems:

          Historical file systems disallow the null byte and the ASCII
          slash character as a part of the file name.

       2) Compatibility with existing programs:

          The existing model for multibyte processing is that ASCII does
          not occur anywhere in a multibyte encoding.  There should be no
          ASCII code values for any part of a transformation format
          representation of a character that was not in the ASCII character
          set in the UCS representation of the character.

       3) Ease of conversion from/to UCS.

       4) The first byte should indicate the number of bytes to follow in a
          multibyte sequence.

       5) The transformation format should not be extravagant in terms of
          number of bytes used for encoding.


   Proposed FSS-UTF
   ----------------

   The proposed UCS transformation format encodes UCS values in the range
   [0,0x7fffffff] using multibyte characters of lengths 1, 2, 3, 4, and 5
   bytes.  For all encodings of more than one byte, the initial byte
   determines the number of bytes used and the high-order bit in each byte
   is set.

   An easy way to remember this transformation format is to note that the
   number of high-order 1's in the first byte is the same as the number of
   subsequent bytes in the multibyte character:

      Bits  Hex Min  Hex Max          Byte Sequence in Binary
   1    7  00000000 0000007f 0zzzzzzz
   2   13  00000080 0000207f 10zzzzzz 1yyyyyyy
   3   19  00002080 0008207f 110zzzzz 1yyyyyyy 1xxxxxxx
   4   25  00082080 0208207f 1110zzzz 1yyyyyyy 1xxxxxxx 1wwwwwww
   5   31  02082080 7fffffff 11110zzz 1yyyyyyy 1xxxxxxx 1wwwwwww 1vvvvvvv

   The bits included in the byte sequence is biased by the minimum value
   so that if all the z's, y's, x's, w's, and v's are zero, the minimum
   value is represented.  In the byte sequences, the lowest-order encoded
   bits are in the last byte; the high-order bits (the z's) are in the
   first byte.

   This transformation format uses the byte values in the entire range of
   0x80 to 0xff, inclusive, as part of multibyte sequences.  Given the
   assumption that at most there are seven (7) useful bits per byte, this
   transformation format is close to minimal in its number of bytes used.

   Below are sample implementations of the C standard wctomb() and
   mbtowc() functions which demonstrate the algorithms for converting from
   UCS to the transformation format and converting from the transformation
   format to UCS.  The sample implementations include error checks, some
   of which may not be necessary for conformance:

  #define OFF1   0x0000080
  #define OFF2   0x0002080
  #define OFF3   0x0082080
  #define OFF4   0x2082080

  int wctomb(char *s, wchar_t wc)
  {
  	if (s == 0)
  		return 0;	/* no shift states */
  #ifdef wchar_t_is_signed
  	if (wc < 0)
  		goto bad;
  #endif
  	if (wc <= 0x7f)		/* fits in 7 bits */
  	{
  		s[0] = wc;
  		return 1;
  	}
  	if (wc <= 0x1fff + OFF1)	/* fits in 13 bits */
  	{
  		wc -= OFF1;
  		s[0] = 0x80 | (wc >> 7);
  		s[1] = 0x80 | (wc & 0x7f);
  		return 2;
  	}
  	if (wc <= 0x7ffff + OFF2)	/* fits in 19 bits */
  	{
  		wc -= OFF2;
  		s[0] = 0xc0 | (wc >> 14);
  		s[1] = 0x80 | ((wc >> 7) & 0x7f);
  		s[2] = 0x80 | (wc & 0x7f);
  		return 3;
  	}
  	if (wc <= 0x1ffffff + OFF3)	/* fits in 25 bits */
  	{
  		wc -= OFF3;
  		s[0] = 0xe0 | (wc >> 21);
  		s[1] = 0x80 | ((wc >> 14) & 0x7f);
  		s[2] = 0x80 | ((wc >> 7) & 0x7f);
  		s[3] = 0x80 | (wc & 0x7f);
  		return 4;
  	}
  #if !defined(wchar_t_is_signed) || defined(wchar_t_is_more_than_32_bits)
  	if (wc > 0x7fffffff)
  		goto bad;
  #endif
```

```
	wc -= OFF4;
	s[0] = 0xf0 | (wc >> 28);
	s[1] = 0x80 | ((wc >> 21) & 0x7f);
	s[2] = 0x80 | ((wc >> 14) & 0x7f);
	s[3] = 0x80 | ((wc >> 7) & 0x7f);
	s[4] = 0x80 | (wc & 0x7f);
	return 5;
bad:;
	errno = EILSEQ;
	return -1;
}


int mbtowc(wchar_t *p, const char *s, size_t n)
{
	unsigned char *uc;	/* so that all bytes are nonnegative */

	if ((uc = (unsigned char *)s) == 0)
		return 0;		/* no shift states */
	if (n == 0)
		return -1;
	if ((*p = uc[0]) < 0x80)
		return uc[0] != '\0';	/* return 0 for '\0', else 1 */
	if (uc[0] < 0xc0)
	{
		if (n < 2)
			return -1;
		if (uc[1] < 0x80)
			goto bad;
		*p &= 0x3f;
		*p <<= 7;
		*p |= uc[1] & 0x7f;
		*p += OFF1;
		return 2;
	}
	if (uc[0] < 0xe0)
	{
		if (n < 3)
			return -1;
		if (uc[1] < 0x80 || uc[2] < 0x80)
			goto bad;
		*p &= 0x1f;
		*p <<= 14;
		*p |= (uc[1] & 0x7f) << 7;
		*p |= uc[2] & 0x7f;
		*p += OFF2;
		return 3;
	}
	if (uc[0] < 0xf0)
	{
		if (n < 4)
			return -1;
		if (uc[1] < 0x80 || uc[2] < 0x80 || uc[3] < 0x80)
			goto bad;
		*p &= 0x0f;
		*p <<= 21;
		*p |= (uc[1] & 0x7f) << 14;
		*p |= (uc[2] & 0x7f) << 7;
		*p |= uc[3] & 0x7f;
		*p += OFF3;
		return 4;
	}
	if (uc[0] < 0xf8)
	{
		if (n < 5)
```

```
				return -1;
			if (uc[1] < 0x80 || uc[2] < 0x80 || uc[3] < 0x80 || uc[4] < 0x80)
				goto bad;
			*p &= 0x07;
			*p <<= 28;
			*p |= (uc[1] & 0x7f) << 21;
			*p |= (uc[2] & 0x7f) << 14;
			*p |= (uc[3] & 0x7f) << 7;
			*p |= uc[4] & 0x7f;
			if (((*p += OFF4) & ~(wchar_t)0x7fffffff) == 0)
				return 5;
	}
bad:;
	errno = EILSEQ;
	return -1;
}
```

We define 7 byte types:

```
T0	0xxxxxxx	7 free bits
Tx	10xxxxxx	6 free bits
T1	110xxxxx	5 free bits
T2	1110xxxx	4 free bits
T3	11110xxx	3 free bits
T4	111110xx	2 free bits
T5	111111xx	2 free bits
```

Encoding is as follows.

```
>From hex	Thru hex	Sequence		Bits
00000000	0000007f	T0			7
00000080	000007FF	T1 Tx			11
00000800	0000FFFF	T2 Tx Tx		16
00010000	001FFFFF	T3 Tx Tx Tx		21
00200000	03FFFFFF	T4 Tx Tx Tx Tx		26
04000000	FFFFFFFF	T5 Tx Tx Tx Tx Tx	32
```

Some notes:

1. The 2 byte sequence has 2^11 codes, yet only 2^11-2^7
are allowed. The codes in the range 0-7f are illegal.
I think this is preferable to a pile of magic additive
constants for no real benefit. Similar comment applies
to all of the longer sequences.

2. The 4, 5, and 6 byte sequences are only there for
political reasons. I would prefer to delete these.

3. The 6 byte sequence covers 32 bits, the FSS-UTF
proposal only covers 31.

4. All of the sequences synchronize on any byte that is
not a Tx byte.