- home
- blog
- files
- contact

Unicode, UTF-8 tutorial

how are you お元気ですか comment allez-vous 어떻게 지내세요 come stai kuinka voit как дела wie geht's dir 您好吗 cómo estás τι κάνεις ce mai faci como vai você как си jak se máš كيف حالك hur mår du jak się masz ??

This tutorial covers Unicode and its UTF-8 mapping standard. It will start at zero with explaining bits and binary structures, followed by an explanation of the ASCII, extended ASCII and Unicode character sets and ending with some conclusions on how to use UTF-8 en-/decoding with Flash & PHP.

What are bytes and bits?

Bit means “binary digit” and is the smallest unit of computerized data. A bit is a 2-base number, i.e. it has either the value of 0 or 1.
A **byte** is an amount of memory, a certain collection of bits, originally variable in size but now almost always eight bits. This makes 2^8 or 256 possible values for a byte.

byte =	1	2	3	4	5	6	7	8
	bit	bit	bit	bit	bit	bit	bit	bit

1|0 1|0 1|0 1|0 1|0 1|0 1|0 1|0

Some example bytes could be 00000001 or 11111111 or 01010011.
Now how can we calculate the decimal value of this binary encoded byte. What we need is a conversion from base 2 to base 10.
Every 1 or 0 of these binary values is associated with an exponential of 2. For 8 bits it looks like the following:

	1	2	3	4	5	6	7	8
byte =	**128**	**64**	**32**	**16**	**8**	**4**	**2**	**1**
	(2^7)	(2^6)	(2^5)	(2^4)	(2^3)	(2^2)	(2^1)	(2^0)
	1|0	1|0	1|0	1|0	1|0	1|0	1|0	1|0

The calculation of the decimal equivalent of the binary value 00000001:

	128	**64**	**32**	**16**	**8**	**4**	**2**	**1**	
byte =	0	0	0	0	0	0	0	1	= 1

The calculation of the decimal equivalent of the binary value 11111111:

	128	**64**	**32**	**16**	**8**	**4**	**2**	**1**	
byte =	1	1	1	1	1	1	1	1	= 128+64+32+16+8+4+2+1 = 255

The calculation of the decimal equivalent of the binary value 01010011:

	128	**64**	**32**	**16**	**8**	**4**	**2**	**1**
byte =								

0 1 0 1 0 0 1 1 = 64+16+2+1 = 83

What is ASCII?

ASCII stands for American Standard Code for Information Interchange and is a standard for assigning numerical values to the set of letters in the Roman alphabet and typographic characters.
The ASCII character set can be represented by 7 bits. This makes 2^7 or 128 different values resp. characters.
As ASCII uses only 7 of the 8 bits available of an byte the first bit is always 0: 0xxxxxxx;
Below there is a table of decimal values, their binary expressions and the character assigned to that value due to the ASCII standard. The first 32 characters are control characters.

dec	bin	char	dec	bin	char
0	00000000	**NUL (null)**	64	01000000	**@**
1	00000001	**SOH (start of heading)**	65	01000001	**A**
2	00000010	**STX (start of text)**	66	01000010	**B**
3	00000011	**ETX (end of text)**	67	01000011	**C**
4	00000100	**EOT (end of transmission)**	68	01000100	**D**
5	00000101	**ENQ (enquiry)**	69	01000101	**E**
6	00000110	**ACK (acknowledge)**	70	01000110	**F**
7	00000111	**BEL (bell)**	71	01000111	**G**
8	00001000	**BS (backspace)**	72	01001000	**H**
9	00001001	**TAB (horizontal tab)**	73	01001001	**I**
10	00001010	**LF (NL line feed, new line)**	74	01001010	**J**

11	00001011	**VT (vertical tab**	75	01001011	**K**
12	00001100	**FF (NP form feed, new page)**	76	01001100	**L**
13	00001101	**CR (carriage return)**	77	01001101	**M**
14	00001110	**SO (shift out)**	78	01001110	**N**
15	00001111	**SI (shift in)**	79	01001111	**O**
16	00010000	**DLE (data link escape)**	80	01010000	**P**
17	00010001	**DC1 (device control 1)**	81	01010001	**Q**
18	00010010	**DC2 (device control 2)**	82	01010010	**R**
19	00010011	**DC3 (device control 3)**	83	01010011	**S**
20	00010100	**DC4 (device control 4)**	84	01010100	**T**
21	00010101	**NAK (negative acknowledge)**	85	01010101	**U**
22	00010110	**SYN (synchronous idle)**	86	01010110	**V**
23	00010111	**ETB (end of trans. block)**	87	01010111	**W**
24	00011000	**CAN (cancel)**	88	01011000	**X**
25	00011001	**EM (end of medium)**	89	01011001	**Y**
26	00011010	**SUB (substitute)**	90	01011010	**Z**
27	00011011	**ESC (escape)**	91	01011011	**[**
28	00011100	**FS (file separator)**	92	01011100	
29	00011101	**GS (group separator)**	93	01011101	**]**
30	00011110	**RS (record separator)**	94	01011110	**^**
31	00011111	**US (unit separator)**	95	01011111	**_**
32	00100000	**SPACE**	96	01100000	**`**
33	00100001	**!**	97	01100001	**a**

34	00100010	**“**	98	01100010	**b**
35	00100011	**#**	99	01100011	**c**
36	00100100	**$**	100	01100100	**d**
37	00100101	**%**	101	01100101	**e**
38	00100110	**&**	102	01100110	**f**
39	00100111	**‘**	103	01100111	**g**
40	00101000	**(**	104	01101000	**h**
41	00101001	**)**	105	01101001	**i**
42	00101010	*****	106	01101010	**j**
43	00101011	**+**	107	01101011	**k**
44	00101100	**,**	108	01101100	**l**
45	00101101	**-**	109	01101101	**m**
46	00101110	**.**	110	01101110	**n**
47	00101111	**/**	111	01101111	**o**
48	00110000	**0**	112	01110000	**p**
49	00110001	**1**	113	01110001	**q**
50	00110010	**2**	114	01110010	**r**
51	00110011	**3**	115	01110011	**s**
52	00110100	**4**	116	01110100	**t**
53	00110101	**5**	117	01110101	**u**
54	00110110	**6**	118	01110110	**v**
55	00110111	**7**	119	01110111	**w**
56	00111000	**8**	120	01111000	**x**

57	00111001	**9**	121	01111001	**y**
58	00111010	**:**	122	01111010	**z**
59	00111011	**;**	123	01111011	**(**
60	00111100	<	124	01111100	|
61	00111101	=	125	01111101	**)**
62	00111110	>	126	01111110	~
63	00111111	**?**	127	01111111	**DEL**

What is extended ASCII?

“Because the number of written symbols used in common natural languages far exceeds the limited range of the ASCII code, many extensions to it have been used to facilitate handling of those languages. Foreign markets for computers and communication equipment were historically open long before standards bodies had time to deliberate upon the best way to accommodate them, so there are many incompatible proprietary extensions to ASCII.

Since ASCII is a 7-bit code, and most computers manipulate data in 8-bit bytes, many extensions use the additional 128 codes available by using all 8 bits of each byte. This helps include many languages otherwise not easily representable in ASCII, but still not enough to cover all languages of countries in which computers are sold, so even these 8-bit extensions had to have local variants.”

“Eventually, ISO released its **ISO 8859** standards describing its own set of 8-bit ASCII extensions. The most popular was **ISO 8859-1**, also called **ISO Latin1**, which contained characters sufficient for the most common Western European languages. Variations were

> standardized for other languages as well: ISO 8859-2 for Eastern European languages and ISO 8859-5 for Cyrillic languages, for example."
> (source link: answers.com)

Often the term ASCII is associated with the 8 bit ISO 8859 standards. For the rest of the document i will refer to the 7 bit standard as ASCII and to the 8 bit ISO 8859 standards as extended ASCII.

What is Unicode?

Unicode is the attempt to create a single unified character set. Unicode is a means to assign a unique number for all characters used by humans in written language. The Unicode standard is basically nothing but tables listing every charcter in written language with its corresponding unique number. One do not need to say that this amount of characters by far exceeds 256 and one byte will not suffice to represent all these characters. Several mechanisms have therefore been suggested to implement Unicode, i.e. to represent these unique numbers as a sequence of bytes.

These mapping methods are called the **UTF (Unicode Transformation Format)**. Among them are **UTF-8, UTF-16, UTF-32**. The numbers indicate the number of bits in one unit. So UTF-8 uses 8 bits per unit. This corresponds to the bits used by one byte as described above and is therefore best processed by our operating system. Hence UTF-8 emerged to the standard encoding for interchange of unicode text with UTF-16 and UTF-32 being used mainly for internal processing.

What is UTF-8?

UTF-8 is a mapping method to represent all Unicode characters as a sequence of bytes.
UTF-8 has the following properties:

- Unicode characters U+0000 (00000000) to U+007F (01111111) are encoded simply as bytes 0×00 to 0×7F (ASCII compatibility). This means that files and strings which contain only 7-bit ASCII characters have the same encoding under both ASCII and UTF-8. This makes UTF-8 a superset of ASCII.
- All Unicode characters >U+007F are encoded as a sequence of several bytes, each of which has the most significant bit set. Therefore, no ASCII byte (0×00-0×7F) can appear as part of any other character.
- The first byte of a multibyte sequence that represents a non-ASCII character is always in the range 0xC0 (11000000) to 0xFD (11111101) and it indicates how many bytes follow for this character. All further bytes in a multibyte sequence are in the range 0×80 (10000000) to 0xBF (10111111). This allows easy resynchronization and makes the encoding stateless and robust against missing bytes.
- All possible 2^{31} Unicode codes can be encoded.
- UTF-8 encoded characters may theoretically be up to six bytes long, however 16-bit BMP characters are only up to three bytes long.
- The sorting order of Bigendian UCS-4 byte strings is preserved.
- The bytes 0xFE (11111110) and 0xFF (11111111) are never used in the UTF-8 encoding.

(source link: www.cl.cam.ac.uk)

So what does all this mean? As we have learned above ASCII characters can be represented by 7 bits. Hence a 8 bit representation of ASCII characters will always have a leading 0 (0xxxxxxx). At this point UTF-8 comes in. UTF-8 encodes all ASCII as is. There will be no change. To indicate that a character is NOT part of the ASCII charset the leading 0 will be changed to an 1. So an UTF-8 character which is not part of ASCII will always start with a 1.

Furthermore the first byte (8 bits) of a byte sequence used to repesent a Unicode character also indicates how many bytes of this sequence will follow. E.g. 110xxxxx: the first 1 indicates that it is not an ASCII character, the second 1 indicates that the next byte belongs to the encoding of this unicode character. 1110xxxx: again the first byte indicates that it is not an ASCII character, the following two 1 indicate that the next two bytes belong to the encoding of this Unicode character. And so on.

The following table makes it clear:

Unicode character number (decimal)	**bit sequence**
U-0 – U-127:	0xxxxxxx (ASCII characters)
U-128 – U-2047:	110xxxxx 10xxxxxx
U-2048 – U-65535:	1110xxxx 10xxxxxx 10xxxxxx
U-65536 – U-2097151:	11110xxx 10xxxxxx 10xxxxxx 10xxxxxx
U-2097152 – U-67108863:	111110xx 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx
U-67108864 – U-2147483647:	1111110x 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx

UTF-8 example

The first thing that becomes clear is that you cannot analyze a string wether it is encoded in UTF-8 or ISO 8859. No check can be done to find out if a string is already UTF-8 encoded or not. Well you can check if the string is valid UTF-8 and hence assume it is UTF-8. A function which does that can be found in the comments to the utf8_encode in the php manual. Validating UTF-8 takes processing

time and the result leaves room for discussion as it could also be an accidently valid ISO 8859 string. This string would mostly consist of some weird signs because then they are all part of the extended ASCII charset (starting with a 1).
But again, you never can be sure.

Let’s make an example:

Take a look at the following string: “© by München”.
München is the german word for Munich. This string contains two characters which are not part of the ASCII charset: the © and the ü.
These characters are part of the extended ASCII charset ISO 8859-1. As this string is not yet encoded to UTF-8 every character is represented by one byte. So if we encode this string to UTF-8 all bytes/characters of the ASCII charset will remain the same, but the © and the ü are not part of ASCII but of extended ASCII and will therefore be transfered to its UTF-8 equivalent.

Let’s take a look at the following table:

character	decimal	binary	UTF-8 binary	UTF-8 decimal	UTF-8 interpreted in ISO 8859-1
®	169	10101001	11000010 10101001	194 169	Â©
ü	252	11111100	11000011 10111100	195 188	Ã¼

The character ü has the ISO 8859-1 value of 252. Transferred to binary this is 11111100. Now when we encode this character to UTF-8 it will be represented by two bytes 11000011 and 10111100 or as decimal values 195 and 188.

Back in ISO 8859-1 the character 195 is interpreted as Ã and the character 188 is interpreted as ¼. Every UTF-8 encoded character can be interpreted by ISO 8859-1. The difference is only the perspective. As in UTF-8 a single character can consist of multiple bytes, in ISO 8859-1 every character is only one byte long. Hence the UTF-8 encoded ü will be interpreted as Ã¼.

You can proove this by looking at the following box: It lists the UTF-8 encoded ü two times. On the left escaped as html entity and on the right the original UTF-8 encoded value. Now change the encoding for this page in your browser to ISO 8859-1 and see what happens. The right part will change to Ã¼.

ü = ü

What happens if you reencode the already UTF-8 encoded string is that the Ã and the ¼ will be interpreted as ISO 8859-1 characters and will be transfered to their UTF-8 equivalents, which will then look like this: ÃƒÂ¼.

Using PHP’s utf8_encode and utf8_decode with Flash

Now let’s get to the question about if and when to use PHP’s UTF-8 functions (see php documentation: utf8_encode ; utf8_decode). The first thing we have learned is that if we have a string containing only ASCII characters, no UTF-8 encoding is needed as ASCII is a subset of UTF-8. The moment characters from the extended ASCII charset are intergrated UTF-8 encoding has to be done.

But what is with decoding from UTF-8. Let's take a closer look on PHP's **utf8_decode** function. The function description in the PHP manual is as follows: "Converts a string with ISO-8859-1 characters encoded with UTF-8 to single-byte ISO-8859-1″
This means that UTF-8 encoded data is transfered to ISO-8859-1. Now you can decide if that's really what you want.

The big advantage of UTF-8 is that it can represent characters which ISO-8859-1 cannot. So decoding to ISO-8859-1 simply means losing these characters. They will be converted to a question mark by the **utf8_decode** function.

What about Flash?

As of version 6, Flash supports Unicode. Flash uses Unicode (if not specified differently with System.useCodepage) as standard for handling character data. The Unicode characters are mapped with UTF-8. (see Macromedia technote ans supportnote on Unicode support)

You often read in tutorials about Flash and the handling of special characters that you should simply use **utf8_encode** on strings before sending them to flash and use **utf8_decode** when retrieving data from flash and everything will be fine.
This is simply not true.

Let's take a deeper look on this case.

Let's assume the charset your server uses is ISO-8859-1. Using **utf8_encode** on your data before sending it to flash is mostly a good idea namely in the case that your data is not already UTF-8 encoded and is containing characters other than ASCII. If you are sending only ASCII characters you don't need to encode the string to UTF-8, but if you do so it will be no problem.

What will happen if you receive data from Flash and use utf8_decode?

All UTF-8 characters will be converted to ISO-8859-1. This means the ü from the example above will be converted to its ISO-8859-1 equivalent.But what if the person sending the data e.g. through an Flash inputfield is from Russia, Japan etc. In this case there will be no ISO-8859-1 equivalent for the UTF-8 characters and the characters get replaced by a question mark. This is apparently not the desired result.

The solution for this is quite simple. If you are planning a flash guestbook, forum, chat etc. which should support all languages just keep your fingers off the **utf8_encode** and **utf8_decode** function. Just save your received data as is to e.g. a database. The result will be some weird signs as they are interpreted as ISO-8859-1. But don’t care. When it’s time to send this data back to flash, get your data from the database and send it as is, because it is already UTF-8 encoded by Flash. All other data which does not originate from Flash, e.g. some text for a button, has to be UTF-8 encoded (again only if it contains characters of the extended ASCII charset) before it is sent to Flash.

What about textfields in Flash?

You must not use textfields with **embedded fonts**. As these textfields will only show the characters you have embedded and you cannot embed the whole Unicode charset. If fonts are **not embedded** Flash will look on the client’s system for the font you chose and will use this font to display the content of the textfield. In this case the restriction is on the font’s supported characters. To display most Unicode characters on Windows systems the font **Arial Unicode MS** is the best choice, as it is Windows standard and supports 51,180 characters.

Another possibility is using Flash’s **device fonts**. Flash’s device fonts are no fonts themselves. A device font defines a general font type. During playback of the movie, these fonts will be substituted with an installed font that matches that type. E.g. the device font _sans will normally be mapped with Arial on Windows systems and Helvetica on a Mac. These mapped fonts normally are standard system fonts with a good chance to support Unicode characters. See this Macromedia technote for further information.

CXS and UTF-8

Now when using a CXS parser on data already UTF-8 encoded you can use the flag **NO_UTF8** to prevent encoding. On the other side when retrieving data from Flash which possibly consists of characters which are not in the ISO-8859-1 charset use the flag **NO_UTF8** to prevent decoding to ISO-8859-1.

Links

Some links with additional information on the topic:

- What is Unicode? by unicode.org
- UTF-8 specification ISO 10646-1:2000 Annex D and RFC 3629
- a good UTF-8 sample site
- unicode tables, an UTF converter etc. are available on this site
- a short and good tutorial on Flash MX & Unicode

This entry was posted on **Saturday, February 12th, 2005** at **16:54**. It is filed under tutorials and tagged with ascii, character set, unicode, utf8. You can follow any responses to this entry through the RSS 2.0 feed. You can leave a response, or trackback from your own site.

One Comment

001
perry - April 1st, 2009 at 4:46 pm

i never post, but had to say thnx to the wrighter, you made me a very happy developper today, i finally get it now…. great job explaining!

Name:	
Mail:	
Website:	
Comment:	
Validation:	B9VQ

Reset Submit Comment

-

- categories

  - flash
  - flex
  - java
  - javascript
  - miscellaneous

  - music
  - php
  - tutorials

- **bookmarks**
  - documentation

- **zehnet.de info**
  - about zehnet.de
  - contact zehnet.de

zehnet.de archive

- November 2010 (1)
- August 2009 (1)
- June 2009 (1)
- May 2009 (1)
- March 2009 (1)
- February 2009 (2)
- April 2008 (1)
- May 2006 (1)
- March 2005 (1)
- February 2005 (1)

meta

- Log in
- Friends Log in
- impressum

zehnet.de | programming talk and more | WordPress//Nova9 parsed this page in 0.174 seconds.
This blog is protected by Dave's Spam Karma 2: 11192 Spams eaten and counting...