

# UTF-8

**UTF-8** is a [variable-width character encoding](#) used for electronic communication. Defined by the Unicode Standard, the name is derived from *Unicode* (or *Universal Coded Character Set*) *Transformation Format* – *8-bit*.<sup>[1]</sup>

UTF-8	
Standard	Unicode Standard ( <a href="http://www.unicode.org/versions/latest/">http://www.unicode.org/versions/latest/</a> ) <span>↗</span>
Classification	Unicode Transformation Format, extended ASCII, variable-width encoding
Extends	US-ASCII
Transforms / Encodes	ISO 10646 (Unicode)
Preceded by	UTF-1
<div><span>↕</span> <span>⌕</span> <span>✎</span> (<a href="https://en.wikipedia.org/w/index.php?title=Template:Infobox_character_encoding&amp;action=edit">https://en.wikipedia.org/w/index.php?title=Template:Infobox_character_encoding&amp;action=edit</a>)</div>	

UTF-8 is capable of encoding all 1,112,064<sup>[nb 1]</sup> valid character [code points](#) in [Unicode](#) using one to four one-[byte](#) (8-bit) code units. Code points with lower numerical values, which tend to occur more frequently, are encoded using fewer bytes. It was designed for [backward compatibility](#) with [ASCII](#): the first 128 characters of Unicode, which correspond one-to-one with ASCII, are encoded using a single byte with the same binary value as ASCII, so that valid ASCII text is valid UTF-8-encoded Unicode as well. Since ASCII bytes do not occur when encoding non-ASCII code points into UTF-8, UTF-8 is safe to use within most programming and document languages that interpret certain ASCII characters in a special way, such as ␣ ([slash](#)) in filenames, ␣ ([backslash](#)) in [escape sequences](#), and ␣ in [printf](#).

UTF-8 was designed as a superior alternative to [UTF-1](#), a proposed variable-width encoding with partial ASCII compatibility which lacked some features including [self-synchronization](#) and fully ASCII-compatible handling of characters such as slashes. [Ken Thompson](#) and [Rob Pike](#) produced the first implementation for the [Plan 9](#) operating system in September 1992.<sup>[2][3]</sup> This led to its adoption by [X/Open](#) as its specification for *FSS-UTF*, which would first be officially presented at [USENIX](#) in January 1993 and subsequently adopted by the [Internet Engineering Task Force](#) (IETF) in RFC 2277 (BCP 18) for future Internet standards work, replacing Single Byte Character Sets such as Latin-1 in older RFCs.

UTF-8 is by far the most common encoding for the [World Wide Web](#), accounting for over 97% of all web pages, and up to 100% for some languages, as of 2021.<sup>[4]</sup>

^ Naming

The official [Internet Assigned Numbers Authority](#) (IANA) code for the encoding is "UTF-8".<sup>[5]</sup> All letters are upper-case, and the name is hyphenated. This spelling is used in all the Unicode Consortium documents relating to the encoding.

Alternatively, the name "**utf-8**" may be used by all standards conforming to the IANA list (which include [CSS](#), [HTML](#), [XML](#), and [HTTP headers](#)),<sup>[6]</sup> as the declaration is case insensitive.<sup>[5]</sup>

Other variants, such as those that omit the hyphen or replace it with a space, i.e. "**utf8**" or "**UTF 8**", are not accepted as correct by the governing standards.<sup>[7]</sup> Despite this, most [web browsers](#) can understand them, and so standards intended to describe existing practice (such as HTML5) may effectively require their recognition.<sup>[8]</sup>

Unofficially, **UTF-8-BOM** and **UTF-8-NOBOM** are sometimes used for text files which contain or don't contain a [byte order mark](#) (BOM), respectively. In Japan especially, UTF-8 encoding without a BOM is sometimes called "**UTF-8N**".<sup>[9][10]</sup>

[Windows XP](#) and later, including all supported Windows versions, have **codepage 65001**, as a synonym for UTF-8 (since [Windows 7](#) support for UTF-8 is better),<sup>[11]</sup> and Microsoft has a script for [Windows 10](#), to enable it by default for its program [Microsoft Notepad](#).<sup>[12]</sup>

In [PCL](#), UTF-8 is called **Symbol-ID "18N"** (PCL supports 183 character encodings, called Symbol Sets, which potentially could be reduced to one, 18N, that is UTF-8).<sup>[13]</sup>

^ Encoding

Since the restriction of the Unicode code-space to 21-bit values in 2003, UTF-8 is defined to encode code points in one to four bytes, depending on the number of significant bits in the numerical value of the code point. The following table shows the structure of the encoding. The x characters are replaced by the bits of the code point.

Code point <-> UTF-8 conversion

First code point	Last code point	Byte 1	Byte 2	Byte 3	Byte 4
U+0000	U+007F	0xxxxxxx			
U+0080	U+07FF	110xxxxx	10xxxxxx		
U+0800	U+FFFF	1110xxxx	10xxxxxx	10xxxxxx	
U+10000	<sup>[nb 2]</sup> U+10FFFF	11110xxx	10xxxxxx	10xxxxxx	10xxxxxx

The first 128 characters (US-ASCII) need one byte. The next 1,920 characters need two bytes to encode, which covers the remainder of almost all [Latin-script alphabets](#), and also [IPA extensions](#), [Greek](#), [Cyrillic](#), [Coptic](#), [Armenian](#), [Hebrew](#), [Arabic](#), [Syriac](#), [Thaana](#) and [N'Ko](#) alphabets, as well as [Combining Diacritical Marks](#). Three bytes are needed for characters in the rest of the [Basic Multilingual Plane](#), which contains virtually all characters in common use,<sup>[14]</sup> including most [Chinese](#), [Japanese](#) and [Korean characters](#). Four bytes are needed for characters in the [other planes of Unicode](#), which include less common [CJK characters](#), various historic scripts, [mathematical symbols](#), and [emoji](#) (pictographic symbols).

A "character" can actually take more than 4 bytes, e.g. an [emoji flag character](#) takes 8 bytes since it's "constructed from a pair of Unicode scalar values".<sup>[15]</sup>

## Examples



Consider the encoding of the [Euro sign](#), €:

1. The Unicode code point for "€" is U+20AC.
2. As this code point lies between U+0800 and U+FFFF, this will take three bytes to encode.
3. [Hexadecimal](#) 20AC is binary **0010 0000 1010 1100**. The two leading zeros are added because a three-byte encoding needs exactly sixteen bits from the code point.
4. Because the encoding will be three bytes long, its leading byte starts with three 1s, then a 0 (1110...)
5. The four most significant bits of the code point are stored in the remaining low order four bits of this byte (1110**0010**), leaving 12 bits of the code point yet to be encoded (...**0000 1010 1100**).
6. All continuation bytes contain exactly six bits from the code point. So the next six bits of the code point are stored in the low order six bits of the next byte, and 10 is stored in the high order two bits to mark it as a continuation byte (so 10**000010**).
7. Finally the last six bits of the code point are stored in the low order six bits of the final byte, and again 10 is stored in the high order two bits (10**101100**).

The three bytes 1110**0010** 10**000010** 10**101100** can be more concisely written in [hexadecimal](#), as **E2 82 AC**.

The following table summarises this conversion, as well as others with different lengths in UTF-8. The colors indicate how bits from the code point are distributed among the UTF-8 bytes. Additional bits added by the UTF-8 encoding process are shown in black.

Examples of UTF-8 encoding

Character		Binary code point	Binary UTF-8	Hex UTF-8
\$	U+0024	010 0100	00100100	24
¢	U+00A2	000 1010 0010	11000010 10100010	C2 A2
₹	U+0939	0000 1001 0011 1001	11100000 10100100 10111001	E0 A4 B9
€	U+20AC	0010 0000 1010 1100	11100010 10000010 10101100	E2 82 AC
한	U+D55C	1101 0101 0101 1100	11101101 10010101 10011100	ED 95 9C
🕒	U+10348	0 0001 0000 0011 0100 1000	11110000 10010000 10001101 10001000	F0 90 8D 88

Octal

UTF-8's use of six bits per byte to represent the actual characters being encoded, means that [octal](#) notation (which uses 3-bit groups) can aid in the comparison of UTF-8 sequences with one another and in manual conversion.<sup>[16]</sup>

Octal code point <-> Octal UTF-8 conversion

First code point	Last code point	Code point	Byte 1	Byte 2	Byte 3	Byte 4
000	177	xxx	xxx			
0200	3777	xxyy	3xx	2yy		
04000	77777	xyyzz	34x	2yy	2zz	
100000	177777	1xyyzz	35x	2yy	2zz	
0200000	4177777	xyyzzww	36x	2yy	2zz	2ww

With octal notation, the arbitrary octal digits, marked with x, y, z or w in the table, will remain unchanged when converting to or from UTF-8.


Example: Á = U+00C1 = 0301 (in octal) is encoded as 303 201 in UTF-8 (C3 81 in hex).  
Example: € = U+20AC = 20254 is encoded as 342 202 254 in UTF-8 (E2 82 AC in hex).


Codepage layout


The following table summarizes usage of UTF-8 *code units* (individual [bytes](#) or [octets](#)) in a *code page* format. The upper half (0\_ to 7\_) is for bytes used only in single-byte codes, so it looks like a normal code page; the lower half is for continuation bytes (8\_ to B\_) and leading bytes (C\_ to F\_), and is explained further in the legend below.


UTF-8


	_0	_1	_2	_3	_4	_5	_6	_7	_8	_9	_A	_B	_C
(1 byte) 0_	NUL 0000	SOH 0001	STX 0002	ETX 0003	EOT 0004	ENQ 0005	ACK 0006	BEL 0007	BS 0008	HT 0009	LF 000A	VT 000B	FF 000C
(1) 1_	DLE 0010	DC1 0011	DC2 0012	DC3 0013	DC4 0014	NAK 0015	SYN 0016	ETB 0017	CAN 0018	EM 0019	SUB 001A	ESC 001B	FS 001C
(1) 2_	SP 0020	! 0021	" 0022	# 0023	\$ 0024	% 0025	& 0026	' 0027	( 0028	) 0029	* 002A	+ 002B	, 002C
(1) 3_	0 0030	1 0031	2 0032	3 0033	4 0034	5 0035	6 0036	7 0037	8 0038	9 0039	: 003A	; 003B	< 003C
(1) 4_	@ 0040	A 0041	B 0042	C 0043	D 0044	E 0045	F 0046	G 0047	H 0048	I 0049	J 004A	K 004B	L 004C
(1) 5_	P 0050	Q 0051	R 0052	S 0053	T 0054	U 0055	V 0056	W 0057	X 0058	Y 0059	Z 005A	[ 005B	\ 005C
(1) 6_	` 0060	a 0061	b 0062	c 0063	d 0064	e 0065	f 0066	g 0067	h 0068	i 0069	j 006A	k 006B	l 006C
(1) 7_	p 0070	q 0071	r 0072	s 0073	t 0074	u 0075	v 0076	w 0077	x 0078	y 0079	z 007A	{ 007B	 007C
8_	• +00	• +01	• +02	• +03	• +04	• +05	• +06	• +07	• +08	• +09	• +0A	• +0B	• +0C
9_	• +10	• +11	• +12	• +13	• +14	• +15	• +16	• +17	• +18	• +19	• +1A	• +1B	• +1C
A_	• +20	• +21	• +22	• +23	• +24	• +25	• +26	• +27	• +28	• +29	• +2A	• +2B	• +2C
B_	• +30	• +31	• +32	• +33	• +34	• +35	• +36	• +37	• +38	• +39	• +3A	• +3B	• +3C
(2) C_	2 0000	2 0040	LATIN 0080	LATIN 00C0	LATIN 0100	LATIN 0140	LATIN 0180	LATIN 01C0	LATIN 0200	IPA 0240	IPA 0280	IPA 02C0	ACCENT 0300
(2) D_	CYRIL 0400	CYRIL 0440	CYRIL 0480	CYRIL 04C0	CYRIL... 0500	ARMENI 0540	HEBREW 0580	HEBREW 05C0	ARABIC 0600	ARABIC 0640	ARABIC 0680	ARABIC 06C0	SYRIAC 0700
(3) E_	INDIC 0800	MISC. 1000	SYMBOL 2000	KANA... 3000	CJK 4000	CJK 5000	CJK 6000	CJK 7000	CJK 8000	CJK 9000	ASIAN A000	HANGUL B000	HANJA C000
(4) F_	SMP... 10000	□ 40000	□ 80000	SSP... C0000	SPUA-B 100000	4 140000	4 180000	4 1C0000	5 200000	5 1000000	5 2000000	5 3000000	6 4000000

 Blue cells are 7-bit (single-byte) sequences. They must not be followed by a continuation byte.<sup>[17]</sup>

 Orange cells with a large dot are a continuation byte.<sup>[18]</sup> The hexadecimal number shown after the + symbol is the value of the 6 bits they add. This character never occurs as the first byte of a multi-byte sequence.

 White cells are the leading bytes for a sequence of multiple bytes,<sup>[19]</sup> the length shown at the left edge of the row. The text shows the Unicode blocks encoded by sequences starting with this byte, and the hexadecimal code point shown in the cell is the lowest character value encoded using that leading byte.

 Red cells must never appear in a valid UTF-8 sequence. The first two red cells (C0 and C1) could be used only for a 2-byte encoding of a 7-bit ASCII character which should be encoded in 1 byte; as described below, such "overlong" sequences are disallowed.<sup>[20]</sup> To understand why this is, consider the character 128, hex 80, binary 1000 0000. To encode it as 2 characters, the low six bits are stored in the second character as 128 itself 10 000000, but the upper two bits are stored in the first character as 110 00010, making the minimum first character C2. The red cells in the F\_ row (F5 to FD) indicate leading bytes of 4-byte or longer sequences that cannot be valid because they would encode code points larger than the U+10FFFF limit of Unicode (a limit derived from the maximum code point encodable in UTF-16<sup>[21]</sup>). FE and FF do not match any allowed character pattern and are therefore not valid start bytes.<sup>[22]</sup>

 Pink cells are the leading bytes for a sequence of multiple bytes, of which some, but not all, possible continuation sequences are valid. E0 and F0 could start overlong encodings, in this case the lowest non-overlong-encoded code point is shown. F4 can start code points greater than U+10FFFF which are invalid. ED can start the encoding of a code point in the range U+D800–U+DFFF; these are invalid since they are reserved for UTF-16 surrogate halves.<sup>[23]</sup>

## Overlong encodings



In principle, it would be possible to inflate the number of bytes in an encoding by padding the code point with leading 0s. To encode the Euro sign € from the above example in four bytes instead of three, it could be padded with leading 0s until it was 21 bits long – 000 000010 000010 101100, and encoded as 11110000 10000010 10000010 10101100 (or F0 82 82 AC in hexadecimal). This is called an *overlong encoding*.

The standard specifies that the correct encoding of a code point uses only the minimum number of bytes required to hold the significant bits of the code point. Longer encodings are called *overlong* and are not valid UTF-8 representations of the code point. This rule maintains a one-to-one correspondence between code points and their valid encodings, so that there is a unique

valid encoding for each code point. This ensures that string comparisons and searches are well-defined.

## Invalid sequences and error handling



Not all sequences of bytes are valid UTF-8. A UTF-8 decoder should be prepared for:

- invalid bytes
- an unexpected continuation byte
- a non-continuation byte before the end of the character
- the string ending before the end of the character (which can happen in simple string truncation)
- an overlong encoding
- a sequence that decodes to an invalid code point

Many of the first UTF-8 decoders would decode these, ignoring incorrect bits and accepting overlong results. Carefully crafted invalid UTF-8 could make them either skip or create ASCII characters such as NUL, slash, or quotes. Invalid UTF-8 has been used to bypass security validations in high-profile products including Microsoft's [IIS](#) web server<sup>[24]</sup> and Apache's Tomcat servlet container.<sup>[25]</sup> RFC 3629 states "Implementations of the decoding algorithm MUST protect against decoding invalid sequences."<sup>[7]</sup> *The Unicode Standard* requires decoders to "...treat any ill-formed code unit sequence as an error condition. This guarantees that it will neither interpret nor emit an ill-formed code unit sequence."

Since RFC 3629 (November 2003), the high and low surrogate halves used by [UTF-16](#) (U+D800 through U+DFFF) and code points not encodable by UTF-16 (those after U+10FFFF) are not legal Unicode values, and their UTF-8 encoding must be treated as an invalid byte sequence. Not decoding unpaired surrogate halves makes it impossible to store invalid UTF-16 (such as Windows filenames or UTF-16 that has been split between the surrogates) as UTF-8,<sup>[26]</sup> while it is possible with [WTF-8](#).

Some implementations of decoders throw exceptions on errors.<sup>[27]</sup> This has the disadvantage that it can turn what would otherwise be harmless errors (such as a "no such file" error) into a [denial of service](#). For instance early versions of Python 3.0 would exit immediately if the command line or [environment variables](#) contained invalid UTF-8.<sup>[28]</sup> An alternative practice is to replace errors with a replacement character. Since Unicode 6<sup>[29]</sup> (October 2010), the standard (chapter 3) has recommended a "best practice" where the error ends as soon as a disallowed byte is encountered. In these decoders E1, A0, C0 is two errors (2 bytes in the first one). This means an error is no more than three bytes long and never contains the start of a valid character,

and there are 21,952 different possible errors.<sup>[30]</sup> The standard also recommends replacing each error with the [replacement character](#) "�" (U+FFFD).

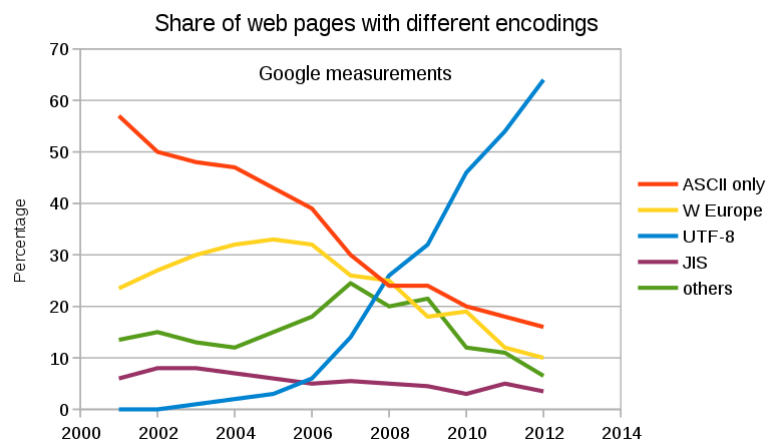
## Byte order mark

If the UTF-16 Unicode [byte order mark](#) (BOM, U+FEFF) character is at the start of a UTF-8 file, the first three bytes will be 0xEF, 0xBB, 0xBF.

The Unicode Standard neither requires nor recommends the use of the BOM for UTF-8, but warns that it may be encountered at the start of a file trans-coded from another encoding.<sup>[31]</sup> While ASCII text encoded using UTF-8 is backward compatible with ASCII, this is not true when Unicode Standard recommendations are ignored and a BOM is added. A BOM can confuse software that isn't prepared for it but can otherwise accept UTF-8, e.g. programming languages that permit non-ASCII bytes in [string literals](#) but not at the start of the file. Nevertheless, there was and still is software that always inserts a BOM when writing UTF-8, and refuses to correctly interpret UTF-8 unless the first character is a BOM (or the file only contains ASCII).

Some programming languages and file formats have their own way of marking usage of encodings like UTF-8 in source code. Examples include [HTML](#) `<meta charset="UTF-8"/>` and [Python 2.7](#) `# coding: utf-8`

## Adoption



Use of the main encodings on the web from 2001 to 2012 as recorded by Google,<sup>[32]</sup> with UTF-8 overtaking all others in 2008 and over 60% of the web in 2012 (since then approaching 100%). The [ASCII](#)-only figure includes all web pages that only contain ASCII characters, regardless of the declared header.

UTF-8 is the recommendation from the [WHATWG](#) for HTML and [DOM](#) specifications,<sup>[33]</sup> and the [Internet Mail Consortium](#) recommends that all e-mail programs be able to display and create mail using UTF-8.<sup>[34][35]</sup> The [World Wide Web Consortium](#) recommends UTF-8 as the default encoding in [XML](#) and [HTML](#) (and not just using UTF-8, also stating it in metadata), "even when all



characters are in the [ASCII](#) range .. Using non-UTF-8 encodings can have unexpected results".<sup>[36]</sup> Many other standards only support UTF-8, e.g. open [JSON](#) exchange requires it.<sup>[37]</sup>

UTF-8 has been the most common encoding for the [World Wide Web](#) since 2008.<sup>[38]</sup> As of August 2021, UTF-8 accounts for on average 97.2% of all web pages; and 984 of the top 1,000 highest ranked web pages.<sup>[4]</sup> UTF-8 includes ASCII as a subset; almost no websites declare only ASCII used.<sup>[39]</sup> Several languages have 100.0% UTF-8 use.

For local text files UTF-8 usage is lower, and many legacy single-byte (and [CJK](#) multi-byte) encodings remain in use. The primary cause is editors that do not display or write UTF-8 unless the first character in a file is a [byte order mark](#) (BOM), making it impossible for other software to use UTF-8 without being rewritten to ignore the byte order mark on input and add it on output.<sup>[40][41]</sup> Recently there has been some improvement, Notepad on [Windows 10](#) writes UTF-8 without a BOM by default,<sup>[42]</sup> and some system files on [Windows 11](#) require UTF-8,<sup>[43]</sup> and almost all files on macOS and Linux are required to be UTF-8 (without a BOM). [Java](#), with its [NIO](#) API defaults to reading and writing UTF-8 files, and is moving to do so for all other file APIs.<sup>[44]</sup> It already does for all APIs on most platforms,<sup>[45]</sup> except [Windows](#). "The choice of UTF-8 applies only to standard Java APIs and not to the Java language, which will continue to use UTF-16."<sup>[44]</sup> Many other programming languages default to UTF-8 for I/O; or plan to migrate to UTF-8, such as [Python](#) which is preparing to "change the default encoding to UTF-8 [since it has] become the de-facto standard text encoding".<sup>[46]</sup> Most databases support UTF-8 (sometimes the only option as with some file formats), including Microsoft's since SQL Server 2019, resulting in 35% speed increase, and "nearly 50% reduction in storage requirements."<sup>[47]</sup>

Internally in software usage is lower, with [UTF-16](#) or [UCS-2](#) in use, particularly on Windows, but also by [JavaScript](#), [Python](#),<sup>[48]</sup> [Qt](#), and many other cross-platform software libraries. Compatibility with the [Windows API](#) is the primary reasons for this, though the belief that direct indexing of BMP improves speed also was a factor. More recent software has started to use UTF-8: the default string primitive used in [Go](#),<sup>[49]</sup> [Julia](#), [Rust](#), [Swift 5](#),<sup>[50]</sup> and [PyPy](#)<sup>[51]</sup> is UTF-8, a future version of Python intends to store strings as UTF-8,<sup>[52]</sup> and modern versions of [Microsoft Visual Studio](#) use UTF-8 internally<sup>[53]</sup> (however still require a command-line switch to read or write UTF-8<sup>[54]</sup>). UTF-8 is the "only text encoding mandated to be supported by the C++ standard", as of [C++20](#).<sup>[55]</sup> As of May 2019, Microsoft reversed its course of only supporting [UTF-16](#) for the Windows API, providing the ability to set UTF-8 as the "code page" for the multi-byte API (previously this was impossible), and now [Microsoft recommends \(programmers\) using UTF-8](#) for [Universal Windows Platform](#) (UWP) apps,<sup>[56]</sup> while continuing to maintain a legacy "Unicode" (meaning UTF-16) interface.<sup>[57]</sup>

## ^ History



The [International Organization for Standardization](#) (ISO) set out to compose a universal multi-byte character set in 1989. The draft ISO 10646 standard contained a non-required [annex](#) called UTF-1 that provided a byte stream encoding of its [32-bit](#) code points. This encoding was not satisfactory on performance grounds, among other problems, and the biggest problem was probably that it did not have a clear separation between ASCII and non-ASCII: new UTF-1 tools would be backward compatible with ASCII-encoded text, but UTF-1-encoded text could confuse existing code expecting ASCII (or [extended ASCII](#)), because it could contain continuation bytes in the range 0x21–0x7E that meant something else in ASCII, e.g., 0x2F for '/', the [Unix path](#) directory separator, and this example is reflected in the name and introductory text of its replacement. The table below was derived from a textual description in the annex.

UTF-1

Number of bytes	First code point	Last code point	Byte 1	Byte 2	Byte 3	Byte 4	Byte 5
1	U+0000	U+009F	00–9F				
2	U+00A0	U+00FF	A0	A0–FF			
2	U+0100	U+4015	A1–F5	21–7E, A0–FF			
3	U+4016	U+38E2D	F6–FB	21–7E, A0–FF	21–7E, A0–FF		
5	U+38E2E	U+7FFFFFFF	FC–FF	21–7E, A0–FF	21–7E, A0–FF	21–7E, A0–FF	21–7E, A0–FF

In July 1992, the [X/Open](#) committee XoJIG was looking for a better encoding. Dave Prosser of [Unix System Laboratories](#) submitted a proposal for one that had faster implementation characteristics and introduced the improvement that 7-bit ASCII characters would only represent themselves; all multi-byte sequences would include only bytes where the high bit was set. The name File System Safe [UCS](#) Transformation Format (FSS-UTF) and most of the text of this proposal were later preserved in the final specification.<sup>[58][59][60][61]</sup>

FSS-UTF



FSS-UTF proposal (1992)

Number of bytes	First code point	Last code point	Byte 1	Byte 2	Byte 3	Byte 4	Byte 5
1	U+0000	U+007F	0xxxxxxx				
2	U+0080	U+207F	10xxxxxx	1xxxxxxx			
3	U+2080	U+8207F	110xxxxx	1xxxxxxx	1xxxxxxx		
4	U+82080	U+208207F	1110xxxx	1xxxxxxx	1xxxxxxx	1xxxxxxx	
5	U+2082080	U+7FFFFFFF	11110xxx	1xxxxxxx	1xxxxxxx	1xxxxxxx	1xxxxxxx

In August 1992, this proposal was circulated by an [IBM X/Open](#) representative to interested parties. A modification by [Ken Thompson](#) of the [Plan 9 operating system](#) group at [Bell Labs](#) made it somewhat less bit-efficient than the previous proposal but crucially allowed it to be [self-synchronizing](#), letting a reader start anywhere and immediately detect byte sequence boundaries. It also abandoned the use of biases and instead added the rule that only the shortest possible encoding is allowed; the additional loss in compactness is relatively insignificant, but readers now have to look out for invalid encodings to avoid reliability and especially security issues. Thompson's design was outlined on September 2, 1992, on a [placemat](#) in a New Jersey diner with [Rob Pike](#). In the following days, Pike and Thompson implemented it and updated [Plan 9](#) to use it throughout, and then communicated their success back to X/Open, which accepted it as the specification for FSS-UTF.<sup>[60]</sup>

FSS-UTF (1992) / UTF-8 (1993)<sup>[2]</sup>

Number of bytes	First code point	Last code point	Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6
1	U+0000	U+007F	0xxxxxxx					
2	U+0080	U+07FF	110xxxxx	10xxxxxx				
3	U+0800	U+FFFF	1110xxxx	10xxxxxx	10xxxxxx			
4	U+10000	U+1FFFFF	11110xxx	10xxxxxx	10xxxxxx	10xxxxxx		
5	U+200000	U+3FFFFFFF	111110xx	10xxxxxx	10xxxxxx	10xxxxxx	10xxxxxx	
6	U+4000000	U+7FFFFFFF	1111110x	10xxxxxx	10xxxxxx	10xxxxxx	10xxxxxx	10xxxxxx

UTF-8 was first officially presented at the [USENIX](#) conference in [San Diego](#), from January 25 to 29, 1993. The [Internet Engineering Task Force](#) adopted UTF-8 in its Policy on Character Sets and Languages in RFC 2277 ([BCP](#) 18) for future Internet standards work, replacing [Single Byte Character Sets](#) such as [Latin-1](#) in older RFCs.<sup>[62]</sup>

In November 2003, UTF-8 was restricted by [RFC 3629](https://datatracker.ietf.org/doc/html/rfc3629) (<https://datatracker.ietf.org/doc/html/rfc3629>)<sup>[62]</sup> to match the constraints of the [UTF-16](#) character encoding: explicitly prohibiting code points corresponding to the high and low surrogate characters removed more than 3% of the three-byte sequences, and ending at U+10FFFF removed more than 48% of the four-byte sequences and all five- and six-byte sequences.

## ^ Standards

There are several current definitions of UTF-8 in various standards documents:

- [RFC 3629](https://datatracker.ietf.org/doc/html/rfc3629) (<https://datatracker.ietf.org/doc/html/rfc3629>)<sup>[63]</sup> / STD 63 (2003), which establishes UTF-8 as a standard Internet protocol element
- [RFC 5198](https://datatracker.ietf.org/doc/html/rfc5198) (<https://datatracker.ietf.org/doc/html/rfc5198>)<sup>[64]</sup> defines UTF-8 [NFC](#) for Network Interchange (2008)
- ISO/IEC 10646:2014 §9.1 (2014)<sup>[63]</sup>
- *The Unicode Standard, Version 11.0* (2018)<sup>[64]</sup>

They supersede the definitions given in the following obsolete works:

- *The Unicode Standard, Version 2.0*, Appendix A (1996)
- ISO/IEC 10646-1:1993 Amendment 2 / Annex R (1996)
- [RFC 2044](https://datatracker.ietf.org/doc/html/rfc2044) (<https://datatracker.ietf.org/doc/html/rfc2044>)<sup>[65]</sup> (1996)
- [RFC 2279](https://datatracker.ietf.org/doc/html/rfc2279) (<https://datatracker.ietf.org/doc/html/rfc2279>)<sup>[66]</sup> (1998)
- *The Unicode Standard, Version 3.0*, §2.3 (2000) plus Corrigendum #1 : UTF-8 Shortest Form (2000)
- *Unicode Standard Annex #27: Unicode 3.1* (2001)<sup>[65]</sup>
- *The Unicode Standard, Version 5.0* (2006)<sup>[66]</sup>
- *The Unicode Standard, Version 6.0* (2010)<sup>[67]</sup>

They are all the same in their general mechanics, with the main differences being on issues such as allowed range of code point values and safe handling of invalid input.

## ^ Comparison with other encodings

Some of the important features of this encoding are as follows:

- *Backward compatibility*: Backward compatibility with ASCII and the enormous amount of software designed to process ASCII-encoded text was the main driving force behind the

design of UTF-8. In UTF-8, single bytes with values in the range of 0 to 127 map directly to Unicode code points in the ASCII range. Single bytes in this range represent characters, as they do in ASCII. Moreover, 7-bit bytes (bytes where the most significant bit is 0) never appear in a multi-byte sequence, and no valid multi-byte sequence decodes to an ASCII code-point. A sequence of 7-bit bytes is both valid ASCII and valid UTF-8, and under either interpretation represents the same sequence of characters. Therefore, the 7-bit bytes in a UTF-8 stream represent all and only the ASCII characters in the stream. Thus, many text processors, parsers, protocols, file formats, text display programs, etc., which use ASCII characters for formatting and control purposes, will continue to work as intended by treating the UTF-8 byte stream as a sequence of single-byte characters, without decoding the multi-byte sequences. ASCII characters on which the processing turns, such as punctuation, whitespace, and control characters will never be encoded as multi-byte sequences. It is therefore safe for such processors to simply ignore or pass-through the multi-byte sequences, without decoding them. For example, ASCII whitespace may be used to [tokenize](#) a UTF-8 stream into words; ASCII line-feeds may be used to split a UTF-8 stream into lines; and ASCII NUL characters can be used to split UTF-8-encoded data into null-terminated strings. Similarly, many format strings used by library functions like "printf" will correctly handle UTF-8-encoded input arguments.

- *Fallback and auto-detection:* Only a small subset of possible byte strings are a valid UTF-8 string: the bytes C0, C1, and F5 through FF cannot appear, and bytes with the high bit set must be in pairs, and other requirements. It is extremely unlikely that a readable text in any [extended ASCII](#) is valid UTF-8. Part of the popularity of UTF-8 is due to it providing a form of backward compatibility for these as well. A UTF-8 processor which erroneously receives extended ASCII as input can thus "auto-detect" this with very high reliability. Fallback errors will be false negatives, and these will be rare. Moreover, in many applications, such as text display, the consequence of incorrect fallback is usually slight. A UTF-8 stream may simply contain errors, resulting in the auto-detection scheme producing false positives; but auto-detection is successful in the majority of cases, especially with longer texts, and is widely used. It also works to "fall back" or replace 8-bit bytes using the appropriate code-point for a legacy encoding only when errors in the UTF-8 are detected, allowing recovery even if UTF-8 and legacy encoding is concatenated in the same file.
- *Prefix code:* The first byte indicates the number of bytes in the sequence. Reading from a stream can instantaneously decode each individual fully received sequence, without first having to wait for either the first byte of a next sequence or an end-of-stream indication. The length of multi-byte sequences is easily determined by humans as it is simply the number of high-order 1s in the leading byte. An incorrect character will not be decoded if a stream ends mid-sequence.
- *Self-synchronization:* The leading bytes and the continuation bytes do not share values (continuation bytes start with the bits 10 while single bytes start with 0 and longer lead bytes

start with 11). This means a search will not accidentally find the sequence for one character starting in the middle of another character. It also means the start of a character can be found from a random position by backing up at most 3 bytes to find the leading byte. An incorrect character will not be decoded if a stream starts mid-sequence, and a shorter sequence will never appear inside a longer one.

- *Sorting order*: The chosen values of the leading bytes means that a list of UTF-8 strings can be sorted in code point order by sorting the corresponding byte sequences.

## Single-byte



- UTF-8 can encode any [Unicode character](#), avoiding the need to figure out and set a "[code page](#)" or otherwise indicate what character set is in use, and allowing output in multiple scripts at the same time. For many scripts there have been more than one single-byte encoding in usage, so even knowing the script was insufficient information to display it correctly.
- The bytes 0xFE and 0xFF do not appear, so a valid UTF-8 stream never matches the UTF-16 [byte order mark](#) and thus cannot be confused with it. The absence of 0xFF (0377) also eliminates the need to escape this byte in [Telnet](#) (and FTP control connection).
- UTF-8 encoded text is larger than specialized single-byte encodings except for plain ASCII characters. In the case of scripts which used 8-bit character sets with non-Latin characters encoded in the upper half (such as most [Cyrillic](#) and [Greek alphabet](#) code pages), characters in UTF-8 will be double the size. For some scripts, such as [Thai](#) and [Devanagari](#) (which is used by various South Asian languages), characters will triple in size. There are even examples where a single byte turns into a composite character in Unicode and is thus six times larger in UTF-8. This has caused objections in India and other countries.
- It is possible in UTF-8 (or any other variable-length encoding) to split or [truncate](#) a string in the middle of a character. If the two pieces are not re-appended later before interpretation as characters, this can introduce an invalid sequence at both the end of the previous section and the start of the next, and some decoders will not preserve these bytes and result in data loss. Because UTF-8 is self-synchronizing this will however never introduce a different valid character, and it is also fairly easy to move the truncation point backward to the start of a character.
- If the code points are all the same size, measurements of a fixed number of them is easy. Due to ASCII-era documentation where "character" is used as a synonym for "byte" this is often considered important. However, by measuring string positions using bytes instead of "characters" most algorithms can be easily and efficiently adapted for UTF-8. Searching for a string within a long string can for example be done byte by byte; the self-synchronization property prevents false positives.

## Other multi-byte





- UTF-8 can encode any [Unicode](#) character. Files in different scripts can be displayed correctly without having to choose the correct code page or font. For instance, Chinese and Arabic can be written in the same file without specialised markup or manual settings that specify an encoding.
- UTF-8 is [self-synchronizing](#): character boundaries are easily identified by scanning for well-defined bit patterns in either direction. If bytes are lost due to error or [corruption](#), one can always locate the next valid character and resume processing. If there is a need to shorten a string to fit a specified field, the previous valid character can easily be found. Many multi-byte encodings such as Shift JIS are much harder to resynchronize. This also means that [byte-oriented string-searching algorithms](#) can be used with UTF-8 (as a character is the same as a "word" made up of that many bytes), optimized versions of byte searches can be much faster due to hardware support and lookup tables that have only 256 entries. Self-synchronization does however require that bits be reserved for these markers in every byte, increasing the size.
- Efficient to encode using simple [bitwise operations](#). UTF-8 does not require slower mathematical operations such as multiplication or division (unlike Shift JIS, [GB 2312](#) and other encodings).
- UTF-8 will take more space than a multi-byte encoding designed for a specific script. East Asian legacy encodings generally used two bytes per character yet take three bytes per character in UTF-8.

## UTF-16



- Byte encodings and UTF-8 are represented by byte arrays in programs, and often nothing needs to be done to a function when converting source code from a byte encoding to UTF-8. [UTF-16](#) is represented by 16-bit word arrays, and converting to UTF-16 while maintaining compatibility with existing [ASCII](#)-based programs (such as was done with Windows) requires every API and data structure that takes a string to be duplicated, one version accepting byte strings and another version accepting UTF-16. If backward compatibility is not needed, all string handling still must be modified.
- Text encoded in UTF-8 will be smaller than the same text encoded in UTF-16 if there are more code points below U+0080 than in the range U+0800..U+FFFF. This is true for all modern European languages. It is often true even for languages like Chinese, due to the large number of spaces, newlines, digits, and HTML markup in typical files.
- Most communication (e.g. HTML and IP) and storage (e.g. for Unix) was designed for a [stream of bytes](#). A UTF-16 string must use a pair of bytes for each code unit:
  - The order of those two bytes becomes an issue and must be specified in the UTF-16 protocol, such as with a [byte order mark](#).
  - If an *odd* number of bytes is missing from UTF-16, the whole rest of the string will be meaningless text. Any bytes missing from UTF-8 will still allow the text to be recovered

accurately starting with the next character after the missing bytes.

## ^ Derivatives



The following implementations show slight differences from the UTF-8 specification. They are incompatible with the UTF-8 specification and may be rejected by conforming UTF-8 applications.

### CESU-8



Unicode Technical Report #26<sup>[68]</sup> assigns the name CESU-8 to a nonstandard variant of UTF-8, in which Unicode characters in [supplementary planes](#) are encoded using six bytes, rather than the four bytes required by UTF-8. CESU-8 encoding treats each half of a four-byte UTF-16 surrogate pair as a two-byte UCS-2 character, yielding two three-byte UTF-8 characters, which together represent the original supplementary character. Unicode characters within the [Basic Multilingual Plane](#) appear as they would normally in UTF-8. The Report was written to acknowledge and formalize the existence of data encoded as CESU-8, despite the [Unicode Consortium](#) discouraging its use, and notes that a possible intentional reason for CESU-8 encoding is preservation of UTF-16 binary collation.

CESU-8 encoding can result from converting UTF-16 data with supplementary characters to UTF-8, using conversion methods that assume UCS-2 data, meaning they are unaware of four-byte UTF-16 supplementary characters. It is primarily an issue on operating systems which extensively use UTF-16 internally, such as [Microsoft Windows](#).

In [Oracle Database](#), the `UTF8` character set uses CESU-8 encoding, and is deprecated. The `AL32UTF8` character set uses standards-compliant UTF-8 encoding, and is preferred.<sup>[69][70]</sup>

CESU-8 is prohibited for use in [HTML5](#) documents.<sup>[71][72][73]</sup>

### MySQL utf8mb3



In [MySQL](#), the `utf8mb3` character set is defined to be UTF-8 encoded data with a maximum of three bytes per character, meaning only Unicode characters in the [Basic Multilingual Plane](#) (i.e. from [UCS-2](#)) are supported. Unicode characters in [supplementary planes](#) are explicitly not supported. `utf8mb3` is deprecated in favor of the `utf8mb4` character set, which uses standards-compliant UTF-8 encoding. `utf8` is an alias for `utf8mb3`, but is intended to become an alias to `utf8mb4` in a future release of MySQL.<sup>[74]</sup> It is possible, though unsupported, to store CESU-8 encoded data in `utf8mb3`, by handling UTF-16 data with supplementary characters as though it is UCS-2.



## Modified UTF-8



*Modified UTF-8* (MUTF-8) originated in the [Java programming language](#). In Modified UTF-8, the [null character](#) (U+0000) uses the two-byte overlong encoding 11000000 10000000 (hexadecimal C0 80), instead of 00000000 (hexadecimal 00).<sup>[75]</sup> Modified UTF-8 strings never contain any actual null bytes but can contain all Unicode code points including U+0000,<sup>[76]</sup> which allows such strings (with a null byte appended) to be processed by traditional [null-terminated string](#) functions. All known Modified UTF-8 implementations also treat the surrogate pairs as in [CESU-8](#).

In normal usage, the language supports standard UTF-8 when reading and writing strings through [InputStreamReader](https://docs.oracle.com/javase/10/docs/api/java/io/InputStreamReader.html) (<https://docs.oracle.com/javase/10/docs/api/java/io/InputStreamReader.html>) and [OutputStreamWriter](https://docs.oracle.com/javase/10/docs/api/java/io/OutputStreamWriter.html) (<https://docs.oracle.com/javase/10/docs/api/java/io/OutputStreamWriter.html>) (if it is the platform's default character set or as requested by the program). However it uses Modified UTF-8 for object [serialization](#)<sup>[77]</sup> among other applications of [DataInput](https://docs.oracle.com/javase/10/docs/api/java/io/DataInput.html) (<https://docs.oracle.com/javase/10/docs/api/java/io/DataInput.html>) and [DataOutput](https://docs.oracle.com/javase/10/docs/api/java/io/DataOutput.html) (<https://docs.oracle.com/javase/10/docs/api/java/io/DataOutput.html>) , for the [Java Native Interface](#),<sup>[78]</sup> and for embedding constant strings in [class files](#).<sup>[79]</sup>

The dex format defined by [Dalvik](#) also uses the same modified UTF-8 to represent string values.<sup>[80]</sup> [Tcl](#) also uses the same modified UTF-8<sup>[81]</sup> as Java for internal representation of Unicode data, but uses strict CESU-8 for external data.

## WTF-8



In WTF-8 (Wobbly Transformation Format, 8-bit) *unpaired* surrogate halves (U+D800 through U+DFFF) are allowed.<sup>[82]</sup> This is necessary to store possibly-invalid UTF-16, such as Windows filenames. Many systems that deal with UTF-8 work this way without considering it a different encoding, as it is simpler.<sup>[83]</sup>

(The term "WTF-8" has also been used humorously to refer to [erroneously doubly-encoded UTF-8](#)<sup>[84][85]</sup> sometimes with the implication that [CP1252](#) bytes are the only ones encoded.)<sup>[86]</sup>

## PEP 383



Version 3 of the [Python programming language](#) treats each byte of an invalid UTF-8 bytestream as an error (see also changes with new UTF-8 mode in Python 3.7<sup>[87]</sup>); this gives 128 different possible errors. Extensions have been created to allow any byte sequence that is assumed to be UTF-8 to be losslessly transformed to UTF-16 or UTF-32, by translating the 128 possible error bytes to reserved code points, and transforming those code points back to error bytes to output UTF-8. The most common approach is to translate the codes to U+DC80...U+DCFF which are low

(trailing) surrogate values and thus "invalid" UTF-16, as used by [Python](#)'s PEP 383 (or "surrogateescape") approach.<sup>[88]</sup> Another encoding called [MirBSD OPTU-8/16](#) converts them to U+EF80...U+FFFF in a [Private Use Area](#).<sup>[89]</sup> In either approach, the byte value is encoded in the low eight bits of the output code point.

These encodings are very useful because they avoid the need to deal with "invalid" byte strings until much later, if at all, and allow "text" and "data" byte arrays to be the same object. If a program wants to use UTF-16 internally these are required to preserve and use filenames that can use invalid UTF-8;<sup>[90]</sup> as the Windows filesystem API uses UTF-16, the need to support invalid UTF-8 is less there.<sup>[88]</sup>

For the encoding to be reversible, the standard UTF-8 encodings of the code points used for erroneous bytes must be considered invalid. This makes the encoding incompatible with WTF-8 or CESU-8 (though only for 128 code points). When re-encoding it is necessary to be careful of sequences of error code points which convert back to valid UTF-8, which may be used by malicious software to get unexpected characters in the output, though this cannot produce ASCII characters so it is considered comparatively safe, since malicious sequences (such as [cross-site scripting](#)) usually rely on ASCII characters.<sup>[90]</sup>

## ^ See also



- [Alt code](#)
- [Character encodings in HTML](#)
- [Comparison of e-mail clients#Features](#)
- [Comparison of Unicode encodings](#)
  - [GB 18030](#)
  - [UTF-EBCDIC](#)
- [Iconv](#)
- [Specials \(Unicode block\)](#)
- [Unicode and email](#)
- [Unicode and HTML](#)
- [Percent-encoding#Current standard](#)

## ^ Notes



1. 17 [planes](#) times 2<sup>16</sup> code points per plane, minus 2<sup>11</sup> technically-invalid [surrogates](#).

2. You might expect larger code points than U+10FFFF to be expressible, but in [RFC3629 §3](#) UTF-8 is limited to match the limits of UTF-16. (As [§12](#) notes, this is changed from RFC 2279.)

## ^ References



1. "Chapter 2. General Structure" (<https://www.unicode.org/versions/Unicode6.0.0/>) . *The Unicode Standard* (6.0 ed.). Mountain View, California, US: The Unicode Consortium. ISBN 978-1-936213-01-6.
2. Pike, Rob (30 April 2003). "UTF-8 history" (<https://www.cl.cam.ac.uk/~mgk25/ucs/utf-8-history.txt>) .
3. Pike, Rob; Thompson, Ken (1993). "Hello World or Καλημέρα κόσμε or こんにちは 世界" (<https://www.cl.cam.ac.uk/~mgk25/ucs/UTF-8-Plan9-paper.pdf>) (PDF). *Proceedings of the Winter 1993 USENIX Conference*.
4. "Usage Survey of Character Encodings broken down by Ranking" ([https://w3techs.com/technologies/cross/character\\_encoding/ranking](https://w3techs.com/technologies/cross/character_encoding/ranking)) . *w3techs.com*. Retrieved 2021-06-30.
5. "Character Sets" (<https://www.iana.org/assignments/character-sets>) . Internet Assigned Numbers Authority. 2013-01-23. Retrieved 2013-02-08.
6. Dürst, Martin. "Setting the HTTP charset parameter" (<https://www.w3.org/International/O-HTTP-char-set>) . W3C. Retrieved 2013-02-08.
7. Yergeau, F. (2003). *UTF-8, a transformation format of ISO 10646* (<https://tools.ietf.org/html/rfc3629>) . Internet Engineering Task Force. doi:10.17487/RFC3629 (<https://doi.org/10.17487%2FRFC3629>) . RFC 3629 (<https://tools.ietf.org/html/rfc3629>) . Retrieved 2015-02-03.
8. "Encoding Standard § 4.2. Names and labels" (<https://encoding.spec.whatwg.org/#names-and-labels>) . WHATWG. Retrieved 2018-04-29.
9. "BOM" (<https://suika.fam.cx/~wakaba/wiki/sw/n/BOM>) . *suikawiki* (in Japanese). Retrieved 2013-04-26.
10. Davis, Mark. "Forms of Unicode" (<https://web.archive.org/web/20050506211548/https://www-128.ibm.com/developerworks/library/utfencodingforms/index.html>) IBM. Archived from the original (<https://www-128.ibm.com/developerworks/library/utfencodingforms/index.html>) on 2005-05-06. Retrieved 2013-09-18.
11. Liviu (2014-02-07). "UTF-8 codepage 65001 in Windows 7 - part I" (<https://www.dostips.com/forum/viewtopic.php?t=5357>) . Retrieved 2018-01-30. "Previously under XP (and, unverified, but probably Vista, too) for loops simply did not work while codepage 65001 was active"
12. "Script How to set default encoding to UTF-8 for notepad by PowerShell" (<https://web.archive.org/web/20201124012523/https://gallery.technet.microsoft.com/scriptcenter/How-to-set-default-2d9669ae>) . *gallery.technet.microsoft.com*. Archived from the original ([https://gallery.technet.microsoft.com/scriptcenter/How-to-set-default-2d9669ae?ranMID=24542&ranEAID=TnL5HPStwNw&ranSiteID=TnL5HPStwNw-1ayuyj6iLWwQHN\\_gl6Np\\_w&tuid=\(1f29517b2ebdfe80772bf649d4c144b1\)\(256380\)\(2459594\)\(TnL5HPStwNw-1ayuyj6iLWwQHN\\_gl6Np\\_w\)](https://gallery.technet.microsoft.com/scriptcenter/How-to-set-default-2d9669ae?ranMID=24542&ranEAID=TnL5HPStwNw&ranSiteID=TnL5HPStwNw-1ayuyj6iLWwQHN_gl6Np_w&tuid=(1f29517b2ebdfe80772bf649d4c144b1)(256380)(2459594)(TnL5HPStwNw-1ayuyj6iLWwQHN_gl6Np_w))) on 2020-11-24. Retrieved 2018-01-30.

13. "HP PCL Symbol Sets | Printer Control Language (PCL & PXL) Support Blog" (<https://web.archive.org/web/20150219212843/http://pclhelp.com/pcl-symbol-sets/>) . 2015-02-19. Archived from the original (<http://pclhelp.com/pcl-symbol-sets/>)  on 2015-02-19. Retrieved 2018-01-30.
14. Allen, Julie D.; Anderson, Deborah; Becker, Joe; Cook, Richard, eds. (2012). "The Unicode Standard, Version 6.1". Mountain View, California: Unicode Consortium.
15. "Apple Developer Documentation" (<https://developer.apple.com/documentation/swift/string>) . *developer.apple.com*. Retrieved 2021-03-15.
16. "BinaryString (flink 1.9-SNAPSHOT API)" (<https://ci.apache.org/projects/flink/flink-docs-release-1.9/api/java/org/apache/flink/table/dataformat/BinaryString.html#compareTo-org.apache.flink.table.dataformat.BinaryString>) . *ci.apache.org*. Retrieved 2021-03-24.
17. "Chapter 3" (<https://www.unicode.org/versions/Unicode13.0.0/ch03.pdf>)  (PDF), *The Unicode Standard*, p. 54
18. "Chapter 3" (<https://www.unicode.org/versions/Unicode13.0.0/ch03.pdf>)  (PDF), *The Unicode Standard*, p. 55
19. "Chapter 3" (<https://www.unicode.org/versions/Unicode13.0.0/ch03.pdf>)  (PDF), *The Unicode Standard*, p. 55
20. "Chapter 3" (<https://www.unicode.org/versions/Unicode13.0.0/ch03.pdf>)  (PDF), *The Unicode Standard*, p. 54
21. Yergeau, F. (November 2003). *UTF-8, a transformation format of ISO 10646* (<https://tools.ietf.org/html/rfc3629>) . IETF. doi:10.17487/RFC3629 (<https://doi.org/10.17487%2FRFC3629>) . STD 63. RFC 3629 (<https://tools.ietf.org/html/rfc3629>) . Retrieved August 20, 2020.
22. "Chapter 3" (<https://www.unicode.org/versions/Unicode13.0.0/ch03.pdf>)  (PDF), *The Unicode Standard*, p. 55
23. Yergeau, F. (November 2003). *UTF-8, a transformation format of ISO 10646* (<https://tools.ietf.org/html/rfc3629>) . IETF. doi:10.17487/RFC3629 (<https://doi.org/10.17487%2FRFC3629>) . STD 63. RFC 3629 (<https://tools.ietf.org/html/rfc3629>) . Retrieved August 20, 2020.
24. Marin, Marvin (2000-10-17). "Web Server Folder Traversal MS00-078" (<https://www.sans.org/resources/malwarefaq/wnt-unicode.php>) .
25. "Summary for CVE-2008-2938" (<https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2008-2938>) . *National Vulnerability Database*.
26. "PEP 529 -- Change Windows filesystem encoding to UTF-8" (<https://www.python.org/dev/peps/pep-0529/>) . *Python.org*. Retrieved 2021-08-27. "This PEP proposes changing the default filesystem encoding on Windows to utf-8, and changing all filesystem functions to use the Unicode APIs for filesystem paths. [...] can correctly round-trip all characters used in paths (on POSIX with surrogateescape handling; on Windows because str maps to the native representation). On Windows bytes cannot round-trip all characters used in paths"
27. "DataInput (Java Platform SE 8)" (<https://docs.oracle.com/javase/8/docs/api/java/io/DataInput.html>) . *docs.oracle.com*. Retrieved 2021-03-24.

28. "Non-decodable Bytes in System Character Interfaces" (<https://www.python.org/dev/peps/pep-0383/>) [🔗](#). *python.org*. 2009-04-22. Retrieved 2014-08-13.
29. "Unicode 6.0.0" (<https://www.unicode.org/versions/Unicode6.0.0/>) [🔗](#).
30. 128 1-byte, (16+5)×64 2-byte, and 5×64×64 3-byte. There may be somewhat fewer if more precise tests are done for each continuation byte.
31. "Chapter 2" (<https://www.unicode.org/versions/Unicode6.0.0/ch02.pdf>) [🔗](#) (PDF), *The Unicode Standard*, p. 30
32. Davis, Mark (2012-02-03). "Unicode over 60 percent of the web" (<https://googleblog.blogspot.com/2012/02/unicode-over-60-percent-of-web.html>) [🔗](#). *Official Google Blog*. Archived (<https://web.archive.org/web/20180809152828/https://googleblog.blogspot.com/2012/02/unicode-over-60-percent-of-web.html>) [🔗](#) from the original on 2018-08-09. Retrieved 2020-07-24.
33. "Encoding Standard" (<https://encoding.spec.whatwg.org/#preface>) [🔗](#). *encoding.spec.whatwg.org*. Retrieved 2020-04-15.
34. "Using International Characters in Internet Mail" (<https://web.archive.org/web/20071026103104/http://www.imc.org/mail-i18n.html>) [🔗](#). Internet Mail Consortium. 1998-08-01. Archived from the original (<https://www.imc.org/mail-i18n.html>) [🔗](#) on 2007-10-26. Retrieved 2007-11-08.
35. "Encoding Standard" (<https://encoding.spec.whatwg.org/#security-background>) [🔗](#). *encoding.spec.whatwg.org*. Retrieved 2018-11-15.
36. "Specifying the document's character encoding" (<https://www.w3.org/TR/html5/document-metadata.html#charset>) [🔗](#). *HTML5.2* (<https://www.w3.org/TR/html5/document-metadata.html>) [🔗](#). World Wide Web Consortium. 14 December 2017. Retrieved 2018-06-03.
37. "The JavaScript Object Notation (JSON) Data Interchange Format" (<https://tools.ietf.org/html/rfc8259>) [🔗](#). IETF. December 2017. Retrieved 16 February 2018.
38. Davis, Mark (2008-05-05). "Moving to Unicode 5.1" (<http://googleblog.blogspot.com/2008/05/moving-to-unicode-51.html>) [🔗](#). Retrieved 2021-02-19.
39. "Usage Statistics and Market Share of US-ASCII for Websites, August 2021" (<https://w3techs.com/technologies/details/en-usascii>) [🔗](#). *w3techs.com*. Retrieved 2020-08-24.
40. "How can I make Notepad to save text in UTF-8 without the BOM?" (<https://stackoverflow.com/questions/8432584/how-can-i-make-notepad-to-save-text-in-utf-8-without-the-bom>) [🔗](#). *Stack Overflow*. Retrieved 2021-03-24.
41. Galloway, Matt. "Character encoding for iOS developers. Or UTF-8 what now?" (<http://www.galloway.me.uk/2012/10/character-encoding-for-ios-developers-utf8/>) [🔗](#). *www.galloway.me.uk*. Retrieved 2021-01-02. "in reality, you usually just assume UTF-8 since that is by far the most common encoding."
42. "Windows 10 Notepad is Getting Better UTF-8 Encoding Support" (<https://www.bleepingcomputer.com/news/microsoft/windows-10-notepad-is-getting-better-utf-8-encoding-support/>) [🔗](#). *BleepingComputer*. Retrieved 2021-03-24. "Microsoft is now defaulting to saving new text files as UTF-8 without BOM as shown below."

43. "Customize the Windows 11 Start menu" (<https://docs.microsoft.com/en-us/windows-hardware/customize/desktop/customize-the-windows-11-start-menu>) docs.microsoft.com. Retrieved 2021-06-29.  
"Make sure your LayoutModification.json uses UTF-8 encoding."
44. "JEP 400: UTF-8 by Default" (<https://openjdk.java.net/jeps/400>) openjdk.java.net. Retrieved 2021-08-24.
45. "Default Java file.encoding and default charset changed to UTF-8" (<https://www.ibm.com/docs/en/i/7.4?topic=jc-default-java-fileencoding-default-charset-changed-utf-8>) www.ibm.com. Retrieved 2021-08-24.
46. "PEP 597 -- Add optional EncodingWarning" (<https://www.python.org/dev/peps/pep-0597/>) Python.org. Retrieved 2021-08-24.
47. "Introducing UTF-8 support for SQL Server" (<https://techcommunity.microsoft.com/t5/sql-server/introducing-utf-8-support-for-sql-server/ba-p/734928>) TECHCOMMUNITY.MICROSOFT.COM. 2019-07-02. Retrieved 2021-08-24. "For example, changing an existing column data type from NCHAR(10) to CHAR(10) using an UTF-8 enabled collation, translates into nearly 50% reduction in storage requirements. [...] In the ASCII range, when doing intensive read/write I/O on UTF-8 , we measured an average 35% performance improvement over UTF-16 using clustered tables with a non-clustered index on the string column, and an average 11% performance improvement over UTF-16 using a heap."
48. Python 2 and early 3 versions on Windows, on Unix it used UTF-32. Newer Python 3 implementations use all three of [ISO-8859-1](#), UCS-2, and UTF-32, depending on the maximum code point needed.
49. "The Go Programming Language Specification" ([https://golang.org/ref/spec#Source\\_code\\_representation](https://golang.org/ref/spec#Source_code_representation)) . Retrieved 2021-02-10.
50. Tsai, Michael J. "Michael Tsai - Blog - UTF-8 String in Swift 5" (<https://mjtsai.com/blog/2019/03/21/utf-8-string-in-swift-5/>) . Retrieved 2021-03-15. "Switching to UTF-8 fulfills one of String's long-term goals to enable high-performance processing, [...] also lays the groundwork for providing even more performant APIs in the future"
51. Mattip (2019-03-24). "PyPy Status Blog: PyPy v7.1 released; now uses utf-8 internally for unicode strings" (<https://morepypy.blogspot.com/2019/03/pypy-v71-released-now-uses-utf-8.html>) PyPy Status Blog. Retrieved 2020-11-21.
52. "PEP 623 -- Remove wstr from Unicode" (<https://www.python.org/dev/peps/pep-0623/>) Python.org. Retrieved 2020-11-21. "Until we drop legacy Unicode object, it is very hard to try other Unicode implementation like UTF-8 based implementation in PyPy"
53. ["/validate-charset \(Validate for compatible characters\)"](#) (<https://docs.microsoft.com/en-us/cpp/build/reference/validate-charset-validate-for-compatible-characters>) docs.microsoft.com. Retrieved 2021-07-19. "Visual Studio uses UTF-8 as the internal character encoding during conversion between the source character set and the execution character set."
54. ["/utf-8 \(Set Source and Executable character sets to UTF-8\)"](#) (<https://docs.microsoft.com/en-us/cpp/build/reference/utf-8-set-source-and-executable-character-sets-to-utf-8>) docs.microsoft.com. Retrieved 2021-07-18.
55. "absent std::u8string in C++11" (<https://newbedev.com/>) NewbeDEV. Retrieved 2021-08-24.



56. "Use the Windows UTF-8 code page - UWP applications" (<https://docs.microsoft.com/en-us/windows/uwp/design/globalizing/use-utf8-code-page>) ↗. *docs.microsoft.com*. Retrieved 2020-06-06. "As of Windows Version 1903 (May 2019 Update), you can use the `ActiveCodePage` property in the `appxmanifest` for packaged apps, or the `fusion manifest` for unpackaged apps, to force a process to use UTF-8 as the process code page. [...] `CP_ACP` equates to `CP_UTF8` only if running on Windows Version 1903 (May 2019 Update) or above and the `ActiveCodePage` property described above is set to UTF-8. Otherwise, it honors the legacy system code page. We recommend using `CP_UTF8` explicitly."
57. "Use the Windows UTF-8 code page" (<https://docs.microsoft.com/en-us/windows/uwp/design/globalizing/use-utf8-code-page>) ↗. *UWP applications*. *docs.microsoft.com*. Retrieved 2020-06-06.
58. "Appendix F. FSS-UTF / File System Safe UCS Transformation format" (<https://www.unicode.org/versions/Unicode1.1.0/appF.pdf>) ↗ (PDF). *The Unicode Standard 1.1*. Archived (<https://web.archive.org/web/20160607215950/https://www.unicode.org/versions/Unicode1.1.0/appF.pdf>) ↗ (PDF) from the original on 2016-06-07. Retrieved 2016-06-07.
59. Whistler, Kenneth (2001-06-12). "FSS-UTF, UTF-2, UTF-8, and UTF-16" (<https://unicode.org/mail-arch/unicode-ml/y2001-m06/0318.html>) ↗. Archived (<https://web.archive.org/web/20160607220249/https://unicode.org/mail-arch/unicode-ml/y2001-m06/0318.html>) ↗ from the original on 2016-06-07. Retrieved 2006-06-07.
60. Pike, Rob (2003-04-30). "UTF-8 history" (<https://www.cl.cam.ac.uk/~mgk25/ucs/utf-8-history.txt>) ↗. Retrieved 2012-09-07.
61. Pike, Rob (2012-09-06). "UTF-8 turned 20 years old yesterday" (<https://plus.google.com/u/0/101960720994009339267/posts/Rz1udTvtiMg>) ↗. Retrieved 2012-09-07.
62. Alvestrand, Harald (January 1998). *IETF Policy on Character Sets and Languages* (<https://tools.ietf.org/html/bcp18>) ↗. doi:10.17487/RFC2277 (<https://doi.org/10.17487%2FRFC2277>) ↗. BCP 18.
63. ISO/IEC 10646:2014 §9.1 ([https://www.iso.org/iso/home/store/catalogue\\_ics/catalogue\\_detail\\_ics.htm?csnumber=63182](https://www.iso.org/iso/home/store/catalogue_ics/catalogue_detail_ics.htm?csnumber=63182)) ↗, 2014.
64. *The Unicode Standard, Version 11.0* (<https://www.unicode.org/versions/Unicode11.0.0/>) ↗ §3.9 D92, §3.10 D95 (<https://www.unicode.org/versions/Unicode11.0.0/ch03.pdf>) ↗, 2018.
65. *Unicode Standard Annex #27: Unicode 3.1* (<https://www.unicode.org/reports/tr27/tr27-3.html>) ↗, 2001.
66. *The Unicode Standard, Version 5.0* (<https://www.unicode.org/versions/Unicode5.0.0/>) ↗ §3.9–§3.10 ch. 3 (<https://www.unicode.org/versions/Unicode5.0.0/ch03.pdf>) ↗, 2006.
67. *The Unicode Standard, Version 6.0* (<https://www.unicode.org/versions/Unicode6.0.0/>) ↗ §3.9 D92, §3.10 D95 (<https://www.unicode.org/versions/Unicode6.0.0/ch03.pdf>) ↗, 2010.
68. McGowan, Rick (2011-12-19). "Compatibility Encoding Scheme for UTF-16: 8-Bit (CESU-8)" (<https://www.unicode.org/reports/tr26/tr26-4.html>) ↗. Unicode Consortium. Unicode Technical Report #26.
69. "Character Set Support" (<https://docs.oracle.com/en/database/oracle/oracle-database/19/sqlrf/Character-Set-Support.html>) ↗. *Oracle Database 19c Documentation, SQL Language Reference*. Oracle Corporation.

70. "Supporting Multilingual Databases with Unicode § Support for the Unicode Standard in Oracle Database" (<https://docs.oracle.com/database/121/NLSPG/ch6unicode.htm#NLSPG-GUID-CD422E4F-C5C6-4E22-B95F-CA9CABBCB543>) ↗. *Database Globalization Support Guide*. Oracle Corporation.
71. "8.2.2.3. Character encodings" (<https://www.w3.org/TR/html51/syntax.html#character-encodings>) ↗. *HTML 5.1 Standard*. W3C.
72. "8.2.2.3. Character encodings" (<https://www.w3.org/TR/html5/syntax.html#character-encodings>) ↗. *HTML 5 Standard*. W3C.
73. "12.2.3.3 Character encodings" (<https://html.spec.whatwg.org/multipage/parsing.html#character-encodings>) ↗. *HTML Living Standard*. WHATWG.
74. "The utf8mb3 Character Set (3-Byte UTF-8 Unicode Encoding)" (<https://dev.mysql.com/doc/refman/8.0/en/charset-unicode-utf8mb3.html>) ↗. *MySQL 8.0 Reference Manual*. Oracle Corporation.
75. "Java SE documentation for Interface java.io.DataInput, subsection on Modified UTF-8" (<https://docs.oracle.com/javase/8/docs/api/java/io/DataInput.html#modified-utf-8>) ↗. Oracle Corporation. 2015. Retrieved 2015-10-16.
76. "The Java Virtual Machine Specification, section 4.4.7: "The CONSTANT\_Utf8\_info Structure" " (<https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-4.html#jvms-4.4.7>) ↗. Oracle Corporation. 2015. Retrieved 2015-10-16.
77. "Java Object Serialization Specification, chapter 6: Object Serialization Stream Protocol, section 2: Stream Elements" (<https://docs.oracle.com/javase/8/docs/platform/serialization/spec/protocol.html#a8299>) ↗. Oracle Corporation. 2010. Retrieved 2015-10-16.
78. "Java Native Interface Specification, chapter 3: JNI Types and Data Structures, section: Modified UTF-8 Strings" ([https://docs.oracle.com/javase/8/docs/technotes/guides/jni/spec/types.html#modified\\_utf\\_8\\_strings](https://docs.oracle.com/javase/8/docs/technotes/guides/jni/spec/types.html#modified_utf_8_strings)) ↗. Oracle Corporation. 2015. Retrieved 2015-10-16.
79. "The Java Virtual Machine Specification, section 4.4.7: "The CONSTANT\_Utf8\_info Structure" " (<https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-4.html#jvms-4.4.7>) ↗. Oracle Corporation. 2015. Retrieved 2015-10-16.
80. "ART and Dalvik" (<https://web.archive.org/web/20130426010617/https://source.android.com/tech/dalvik/dex-format.html>) ↗. *Android Open Source Project*. Archived from the original (<https://source.android.com/tech/dalvik/dex-format.html>) ↗ on 2013-04-26. Retrieved 2013-04-09.
81. "Tcler's Wiki: UTF-8 bit by bit (Revision 6)" ([https://wiki.tcl.tk/\\_/revision?N=1211&V=6](https://wiki.tcl.tk/_/revision?N=1211&V=6)) ↗. 2009-04-25. Retrieved 2009-05-22.
82. Sapin, Simon (2016-03-11) [2014-09-25]. "The WTF-8 encoding" (<https://simonsapin.github.io/wtf-8/>) ↗. Archived (<https://web.archive.org/web/20160524180037/https://simonsapin.github.io/wtf-8/>) ↗ from the original on 2016-05-24. Retrieved 2016-05-24.
83. Sapin, Simon (2015-03-25) [2014-09-25]. "The WTF-8 encoding § Motivation" (<https://simonsapin.github.io/wtf-8/#motivation>) ↗. Archived (<https://github.com/SimonSapin/wtf-8/commit/8f90eccf94057d0e91ce61b7133ace32c33c6085>) ↗ from the original on 2016-05-24. Retrieved 2020-08-26.
84. "WTF-8.com" (<http://wtf-8.com/>) ↗. 2006-05-18. Retrieved 2016-06-21.



85. Speer, Robyn (2015-05-21). "ftfy (fixes text for you) 4.0: changing less and fixing more" (<https://web.archive.org/web/20150530150039/https://blog.luminoso.com/2015/05/21/ftfy-fixes-text-for-you-4-0-changing-less-and-fixing-more/>) . Archived from the original (<https://blog.luminoso.com/2015/05/21/ftfy-fixes-text-for-you-4-0-changing-less-and-fixing-more/>)  on 2015-05-30. Retrieved 2016-06-21.
86. "WTF-8, a transformation format of code page 1252" (<https://web.archive.org/web/20161013072641/http://www-uxsup.csx.cam.ac.uk/~fanf2/hermes/doc/qsmtip/draft-fanf-wtf8.html>) . Archived from the original (<http://www-uxsup.csx.cam.ac.uk/~fanf2/hermes/doc/qsmtip/draft-fanf-wtf8.html>)  on 2016-10-13. Retrieved 2016-10-12.
87. "PEP 540 -- Add a new UTF-8 Mode" (<https://www.python.org/dev/peps/pep-0540/>) . *Python.org*. Retrieved 2021-03-24.
88. von Löwis, Martin (2009-04-22). "Non-decodable Bytes in System Character Interfaces" (<https://www.python.org/dev/peps/pep-0383/>) . Python Software Foundation. PEP 383.
89. "RTFM optu8to16(3), optu8to16vis(3)" (<https://www.mirbsd.org/htman/i386/man3/optu8to16.htm>) . *www.mirbsd.org*.
90. Davis, Mark; Suignard, Michel (2014). "3.7 Enabling Lossless Conversion" (<https://www.unicode.org/reports/tr36/#EnablingLosslessConversion>) . *Unicode Security Considerations*. Unicode Technical Report #36.

## ^ External links



- Original UTF-8 paper ([http://doc.cat-v.org/plan\\_9/4th\\_edition/papers/utf](http://doc.cat-v.org/plan_9/4th_edition/papers/utf))  (or pdf (<https://web.archive.org/web/20000917055036/http://plan9.bell-labs.com/sys/doc/utf.pdf>) ) for Plan 9 from Bell Labs
- UTF-8 test pages:
  - Andreas Prilop (<http://www.user.uni-hannover.de/nhtcapri/multilingual1.html>)
  - Jost Gippert (<http://titus.uni-frankfurt.de/indexe.htm?/unicode/unitest.htm>)
  - World Wide Web Consortium (<http://www.w3.org/2001/06/utf-8-test/UTF-8-demo.html>)
- Unix/Linux: UTF-8/Unicode FAQ (<http://www.cl.cam.ac.uk/~mgk25/unicode.html>) , Linux Unicode HOWTO (<http://www.tldp.org/HOWTO/Unicode-HOWTO.html>) , UTF-8 and Gentoo (<https://wiki.gentoo.org/wiki/UTF-8>)
- Characters, Symbols and the Unicode Miracle (<https://www.youtube.com/watch?v=MijmeoH9LT4>)  on YouTube