Unicode encodings

UTF-8

UTF-8 is a multibyte encoding able to encode the whole Unicode charset. An encoded character takes between 1 and 4 bytes. UTF-8 encoding supports longer byte sequences, up to 6 bytes, but the biggest code point of Unicode 6.0 (U+10FFFF) only takes 4 bytes.

> sequences if it is useless ?

It is possible to be sure that a byte string <bytes> is encoded to UTF-8, because UTF-8 adds markers to each byte. For the first byte of a multibyte character, bit 7 and bit 6 are set (`0b11xxxxxx`); the next bytes have bit 7 set and bit 6 unset (`0b10xxxxxx`).

Another cool feature of UTF-8 is that it has no endianness (it can be read in big or little endian order, it does not matter). Another advantage of UTF-8 is that most C <c> bytes functions are compatible with UTF-8 encoded strings (e.g. :cstrcat or :cprintf), whereas they fail with UTF-16 and UTF-32 encoded strings because these encodings encode small codes with nul bytes.

The problem with UTF-8, if you compare it to ASCII or ISO-8859-1, is that it is a multibyte encoding: you cannot access a character by its character index directly, you have to iterate on each character because each character may have a different length in bytes. If getting a character by its index is a common operation in your program, use a character string <str> instead of a UTF-8 encoded string <bytes>.

> vue grammatical :
>
> "It is possible to be sure that a byte string is encoded by UTF-8, because UTF-8 adds markers to each byte." => "Thanks to markers placed at each byte, it is possible to make sure a byte string is encoded in UTF-8"

> expliquer avant ce que c'était. Considères tu que le lecteur connaît ?

UCS-2, UCS-4, UTF-16 and UTF-32

UCS-2 and **UCS-4** encodings encode <encode> each code point to exactly one unit of, respectivelly, 16 and 32 bits. UCS-4 is able to encode all Unicode 6.0 code points, whereas UCS-2 is limited to BMP <bmp> characters. These encodings are practical because the length in units is the number of characters.

UTF-16 and **UTF-32** encodings use, respectively, 16 and 32 bits units. UTF-16 encodes code points bigger than U+FFFF using two units: a surrogate pair <surrogates>. UCS-2 can be decoded <decode> from UTF-16. UTF-32 is also supposed to use more than one unit for big code points, but in practice, it only requires one unit to store all code points of Unicode 6.0. That's why UTF-32 and UCS-4 are the same encoding.

Encoding	Word size	Unicode support
UCS-2	16 bits	BMP only
UTF-16	16 bits	Full
UCS-4	32 bits	Full
UTF-32	32 bits	Full

Windows 95 <win> uses UCS-2, whereas Windows 2000 uses UTF-16.

> **note**
>
> UCS stands for *Universal Character Set*, and UTF stands for *UCS Transformation format*.

UTF-7

The UTF-7 encoding is similar to the UTF-8 encoding <utf8>, except that it uses 7 bits units instead of 8 bits units. It is used for example in emails with server which are not "8 bits clean".

Byte order marks (BOM)

UTF-16 <utf16> and UTF-32 <utf32> use units bigger than 8 bits, and so are sensitive to endianness. A single unit can be stored as big endian (most significant bits first) or little endian (less significant bits first). BOM is a short byte sequence to indicate the encoding and the endian. It's the U+FEFF code point encoded with the given UTF encoding.

Unicode defines 6 different BOM:

BOM	Encoding	Endian
`0x2B 0x2F 0x76 0x38 0x2D` (5 bytes)	UTF-7 <utf7>	*endianless*
`0xEF 0xBB 0xBF` (3)	UTF-8 <utf8>	*endianless*
`0xFF 0xFE` (2)	UTF-16-LE <utf16>	little endian
`0xFE 0xFF` (2)	UTF-16-BE <utf16>	big endian
`0xFF 0xFE 0x00 0x00` (4)	UTF-32-LE <utf32>	little endian
`0x00 0x00 0xFE 0xFF` (4)	UTF-32-BE <utf32>	big endian

UTF-32-LE BOMs starts with UTF-16-LE BOM.

"UTF-16" and "UTF-32" encoding names are imprecise: depending of the context, format or protocol, it means UTF-16 and UTF-32 with BOM markers, or UTF-16 and UTF-32 in the host endian without BOM. On Windows, "UTF-16" usually means UTF-16-LE.

Some Windows applications, like notepad.exe, use UTF-8 BOM, whereas many applications are unable to detect the BOM, and so the BOM causes trouble. UTF-8 BOM should not be used for better interoperability.

UTF-16 surrogate pairs

Surrogates are characters in the Unicode range U+D800—U+DFFF (2,048 code points): it is also the Unicode category <unicode categories> "surrogate" (Cs). The range is composed of two parts:

- U+D800—U+DBFF (1,024 code points): high surrogates
- U+DC00—U+DFFF (1,024 code points): low surrogates

In UTF-16 <utf16>, characters in ranges U+0000—U+D7FF and U+E000—U+FFFD are stored as a single 16 bits unit. Non-BMP <bmp> characters

(range U+10000—U+10FFFF) are stored as "surrogate pairs", two 16 bits units: an high surrogate (in range U+D800—U+DBFF) followed by a low surrogate (in range U+DC00—U+DFFF). A lone surrogate character is invalid in UTF-16, surrogate characters are always written as pairs (high followed by low).

Examples of surrogate pairs:

Character	Surrogate pair
U+10000	{U+D800, U+DC00}
U+10E6D	{U+D803, U+DE6D}
U+1D11E	{U+D834, U+DD1E}
U+10FFFF	{U+DBFF, U+DFFF}

note

U+10FFFF is the highest code point encodable to UTF-16 and the highest code point of the Unicode Character Set <ucs> 6.0. The {U+DBFF, U+DFFF} surrogate pair is the last available pair.

An UTF-8 <utf8> or UTF-32 <utf32> encoder should not encode surrogate characters (U+D800—U+DFFF), see Non-strict UTF-8 decoder <strict utf8 decoder>.

C <c> functions to create a surrogate pair (encode <encode> to UTF-16) and to join a surrogate pair (decode <decode> from UTF-16):

```
#include <stdint.h>

void
encode_utf16_pair(uint32_t character, uint16_t *units)
{
    unsigned int code;
    assert(0x10000 <= character && character <= 0x10FFFF);
    code = (character - 0x10000);
    units[0] = 0xD800 | (code >> 10);
    units[1] = 0xDC00 | (code & 0x3FF);
}

uint32_t
decode_utf16_pair(uint16_t *units)
```

```
{
    uint32_t code;
    assert(0xD800 <= units[0] && units[0] <= 0xDBFF);
    assert(0xDC00 <= units[1] && units[1] <= 0xDFFF);
    code = 0x10000;
    code += (units[0] & 0x03FF) << 10;
    code += (units[1] & 0x03FF);
    return code;
}
```