Contents

1 Introduction

All data in computers is stored in the format of bits – sequences of zeros and ones. However, when we look at a computer screen, we see digits, letters, images – something we can interpret and understand. To represent data in a human-readable format, computers utilize **encodings**. Regarding characters, encoding is a rule that tells a computer how to map a sequence of bits to a symbol, as well as how to transform a symbol into bits.

Initially, one-byte encodings were widely used. To work with a particular language, computers stored a corresponding encoding and used it to transform bytes representing characters of this language into readable text. This approach, however, had some disadvantages:

1. The maximum number of symbols computers could work with was limited by 256. Moreover, 128 of them were reserved to encode Latin and control characters. The other half was used to encode the symbols of a national alphabet and pseudographics. Due to this limitation, one-byte encodings allowed only bilingual but not multilingual text processing.

2. The set of symbols covered by two encodings could be different. Thus, the transformation between them could lead to losses since symbols missed in one encoding had to be substituted by similar symbols from the other.

3. When passed between devices with different operating systems, data was likely to be corrupted. A converter or additional fonts were required.

4. Some writing systems are impossible to represent using one-byte encodings. An example is hieroglyphic writing.

2 What is Unicode

Unicode is a character encoding standard developed to address the described issues. The standard contains a collection of abstract characters covering all the world's living languages. Each character is assigned to a unique number called **code-point**, which is written in the hexadecimal format preceded by the U+ prefix. For example, the code-point of a Latin capital letter A is U+0041, the code-point of Z is U+005A.

Using the standard, any text can be encoded as a sequence of code points. The table below shows an example of such encoding for the text "Hooray!":

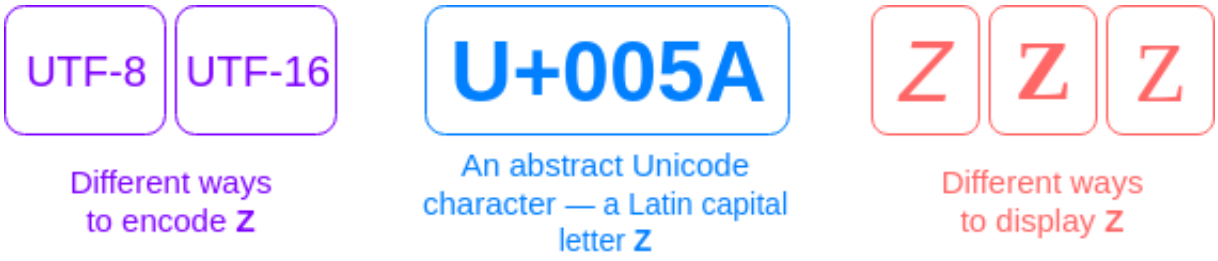
Unicode character	H	o	o	r	a	y	!
Code point	U+0048	U+006F	U+006F	U+0072	U+0061	U+0079	U+0021

Similarly, this sequence of code points can be decoded into the initial Unicode characters.

The total number of code points in Unicode is 1 114 112 ranging from 0 to 10FFFF. According to the 12.1 version of the standard, 137 929 of them are associated with some characters. For convenience, all the code points are divided into 17 groups called **planes**. The most used one is the **Basic Multilingual Plane** (BMP), that includes all Unicode symbols with code points from U+0000 to U+FFFF and covers characters for almost all modern languages.

The term "character" is usually understood as something that represents a letter, a digit or a punctuation mark. However, some Unicode characters correspond to other special symbols. For example, U+0000 is a null character, U+200F is a marker that specifies the writing direction. Also, there is a special range of code points S = [U+D800, U+DFFF] that don't encode any characters. Looking ahead, they are used to encode 4 bytes characters in UTF-16.

Note that we mentioned neither about how Unicode characters are represented in computer memory nor how they are displayed on a screen. In the Unicode standard, the ways how characters are assigned to code points, how code points are transformed into a sequence of bytes and how characters are displayed, are separated from each other. The figure below illustrates this principle:



There is only one Latin capital letter Z with a unique code-point U+005A. Yet, there are several ways to draw the letter on a screen, as well a few ways to transform it into a sequence of bytes. Two widely used transformations are **UTF-8** and **UTF-16**, which are the subject to discuss next.

3 Unicode character encodings

3.1 UTF-8

3.1.1 Description

UTF-8 (8-bit Unicode Transformation Format) is one of the encodings for transforming code points into a sequence of bytes. The encoding has a variable size: depending on the code point, a character

might require from 1 to 4 bytes to be encoded. The smaller the code point, the fewer number of bytes is needed. The following table shows the encoding structure:

Code point range	Bits for code point	Byte 1	Byte 2	Byte 3	Byte 4
U+0000 - U+007F	7	0xxxxxxx	-	-	-
U+0080 - U+07FF	11	110xxxxx	10xxxxxx	-	-
U+0800 - U+FFFF	16	1110xxxx	10xxxxxx	10xxxxxx	-
U+10000 - U+10FFFF	21	11110xxx	10xxxxxx	10xxxxxx	10xxxxxx

The first column represents four ranges that cover all the code points. The second column shows the maximum number of bits a code point from the corresponding range may contain. The rest four columns describe the binary representation of encoded code point. The first byte is called a **leading byte**, each of the rest is a **continuation byte**.

If a character needs only one byte to be encoded, a high bit of this byte is set to zero. Otherwise, first high bits of a leading byte store the number of bytes required to encode the character: 110, 1110, 11110 for 2, 3 and 4 bytes accordingly. The format of a leading byte guarantees that if we sort symbols encoded using UTF-8, we will get the same order as if we sort the symbols' code points.

Two high bits of continuation bytes are always set to 10. This provides a so-called **self-synchronization** property: a search never finds a character starting in the middle of another one. In other words, the start of each character can always be uniquely defined by moving back up to 3 positions to find a leading byte.

To encode a code point, we need to identify the range the code point belongs to, then transform it to the binary format and substitute the "x" characters of the corresponding scheme by the obtained bits. If the number of bits is smaller than specified in the second column, unused "x" characters are filled with zeros.

3.1.2 Example

As an example, let's consider how to encode the square root character ($\surd$) with U+221A code point. The binary representation of the code point is 100010 00011010. Since the code point belongs to the third range, we need 3 bytes to encode it. Below is how encoding looks like:

11100010 10001000 10011010

The bits of the character are shown in blue. The bits shown in red correspond to high bits of the leading and continuation bytes. Since the number of bits for the 3rd scheme is 16, we added two additional bits to the binary representation. These bits are shown in black.

3.2 UTF-16

3.2.1 Description

UTF-16 (16-bit Unicode Transformation Format) is one more approach to encode Unicode characters. Similarly to UTF-8, it has a variable size but uses either one or two 16-bit sequences for encoding, thus takes either two or four bytes. In UTF-16, 16-bit sequences are called **code units**.

The number of code units required to encode a code point depends on the range the code point belongs to. If a code point belongs either to [U+0000, U+D7FF] or [U+E000, U+FFFF], then it needs one code unit. Note that these ranges cover all the BMP characters, thus 2 bytes is always enough to encode them using UTF-16.

The code points in between the two ranges, that is, ones in S = [U+D800, U+DFFF] don't encode any Unicode characters. They are reserved specifically for UTF-16 for the case when two code units are needed, that is, if a code point is in the range [U+010000, U+10FFFF]. To encode a code point from this range, we need to represent it as a pair of two code points from S. Given a code point U, this can be done as follows:

1. Calculate U' = U - 0x10000. Since U is not greater than 0x10FFFF, U' will be a 20-bit number in the range [0x00000, 0xFFFFF].
2. Take the first ten bits of U' and add 0xD800 to them. The result will be the first code unit called **high surrogate**.
3. Take the last ten bits of U' and add 0xDC00 to them. The result will be the second code unit called **low surrogate**.

The first ten bits of U' are always in the range [0x000, 0x4FF], thus, after adding 0xD800 we will get a number in the range [0xD800, 0xDBFF]. Similarly, the last ten bits of U' are always in the range [0x000, 0x4FF], thus, the result of adding 0xDC00 will always be in the range [0xDC00, 0xDFFF]. Therefore, each of the two surrogates will be in the range S = [0xD800, 0xDFFF], as required.

Note that the ranges of surrogates, BMP and all the other Unicode characters don't intersect. Thus, it is always possible to uniquely define the encoding structure by checking to which range the first byte belongs to.

3.2.2 Example

Let's consider how U+1F745 code point is represented in UTF-16. First, we need to subtract 0x10000 from the code-point:

$$0x1F745 - 0x10000 = 0x0F745.$$

Secondly, we need to shift the obtained number by 10 bits to the right to get the first 10 bits. This can be done by dividing the number by 0x400. After that, we need to add D800 to the result to get two bytes of a high surrogate:

$$0x0F745 \ / \ 0x400 + D800 = D83D.$$

Finally, we need to get the first 10 bits and add DC00 to them to get a low surrogate. The first 10 bits can be obtained by finding a remainder from division to 0x400. Thus,

$$0x0F745 \text{ div } 0x400 + DC00 = DF45.$$

The table below represents the obtained encoding:

Format	**High surrogate**		**Low surrogate**	
	Byte 1	Byte2	Byte 3	Byte 4
Hex	D8	3D	DF	45
Binary	11011000	00111101	11011111	01000101

3.2.3 About endianness

The order of bytes within a multi-byte number might be not the same on different computer architectures. In a **big-endian** order, the most significant byte comes first, followed by the least significant one. A **little-endian** order is the opposite. For example, a big-endian order for the bytes of a hex number 0xD83D is (D8, 3D), while a little-endian order is (3D, D8).

To correctly display text encoded using UTF-16, we need to know the type of endianness used for the encoding. Unicode provides several ways to specify it:

- Using **Byte Order Mark** (BOM) – a code point with a value U+FEFF that precedes the first encoded character. The type of endianness is determined by checking the byte order of the value.
- Specifying the byte order explicitly in the title: UTF-16**BE** or UTF-16**LE** for big-endian and little-endian accordingly.

Taking endianness into account, the complete encoding of the U+1F745 code point is as follows:

Encoding	**Format**	**High surrogate**		**Low surrogate**	
		Byte 1	Byte 2	Byte 3	Byte 4
UTF-16BE	Hex	D8	3D	DF	45
	Binary	11011000	00111101	11011111	01000101
UTF-16LE	Hex	3D	D8	45	DF
	Binary	00111101	11011000	01000101	11011111

4 Comparing UTF-8 and UTF-16

4.1 Compatibility

Before Unicode was developed, one of the widely used encodings was **ASCII** (American Standard Code for Information Interchange). This is a one-byte encoding, that originally used only 7 bits to encode the English alphabet, digits, punctuations and control codes. Later, around the world, there emerged extended 8-bit ASCII versions that used the additional 128 values to encode characters of local languages. Since then, a lot of ASCII encoded files were created.

An important property of UTF-8 is that it maintains backward compatibility with ASCII, while UTF-16 does not. Thus, if you open an ASCII encoded file and convert a sequence of bytes into UTF-8, you will get the same file. The compatibility is achieved since the first 128 characters are the same both in Unicode and ASCII, and the corresponding binary codes are equal as well.

4.2 Storage efficiency

Depending on the range code points of Unicode characters belong to, UTF-8 and UTF-16 might require a different amount of memory to store these characters. For example, UTF-8 requires one byte to encode the first 128 Unicode characters, while UTF-16 needs two bytes. The next 1920 characters, that cover almost all Latin-script alphabets, require 16 bits for encoding both in UTF-8 and UTF-16. For the remainder of the BMP characters, that cover most of the world's living languages, UTF-8 needs 24 bits, while UTF-16 needs 16. The remaining range requires 32 bits for both encodings.

If the storage efficiency matters and you know beforehand which type of text you will work with, you may choose the encoding that will require less memory to store data. For example, if you work

with a file containing mostly ASCII characters, UTF-8 will be more efficient than UTF-16. If a file consists of many BMP characters with large code points values, UTF-16 is a better choice than UTF-8.

4.3 Processing efficiency

Depending on the encoding, text processing operations might have different time complexities. For example, if characters of some text have a variable byte length, we need to perform a linear search to find the n-th character in this text. However, if we know that all characters have the same length, such a search can be performed in constant time.

UTF-16 always uses 2 bytes to encode BMP characters, while UTF-8 might require from 1 to 3 bytes. Thus, if you work only with BMP characters and searching is a frequently-used operation, UTF-16 is a better choice in this case.

5 Summary

- Unicode is an encoding standard that introduces a collection of abstract characters covering almost all the world's existing languages. Each character is associated with a unique number called code point. There are several formats that allow representing a code point as a sequence of bytes, as well as several ways to display a character on a screen. In Unicode, the correspondence between characters and code points, transformations formats and fonts do not depend on each other.
- Unicode provides a few formats to encode code points as a sequence of bytes, UTF-8 and UTF-16 being most widely used. Both formats have a variable size: UTF-8 might require from 1 to 4 bytes to encode a code point, while UTF-16 needs either 2 or 4.
- Each of the transformation formats has pros and cons:
  - UTF-8 is compatible with ASCII and requires less memory to store ASCII characters comparing with UTF-16;
  - On the other hand, UTF-16 needs less memory to store BMP characters with large values;
  - UTF-16 always uses 2 bytes to encode BMP characters, while UTF-8 might require from 1 to 3. Thus, BMP characters are more convenient and faster to process using UTF-16.

6 References

- https://en.wikipedia.org/wiki/Unicode
- https://en.wikipedia.org/wiki/UTF-8
- https://en.wikipedia.org/wiki/UTF-16
- https://en.wikipedia.org/wiki/Comparison_of_Unicode_encodings
- https://en.wikipedia.org/wiki/Plane_(Unicode)
- https://en.wikipedia.org/wiki/Endianness
- https://home.unicode.org/
- https://unicode-table.com/en/

- https://en.wikipedia.org/wiki/ASCII
- https://en.wikipedia.org/wiki/Extended_ASCII
- https://habr.com/en/post/312642/
- https://habr.com/en/post/158639/