Seminar paper: Why became Unicode the standard in the IT world and what are the benefits?

Author: Thomas Dauner
Date: 03.01.2022

Abstract

This seminar paper explains the advantages of Unicode and why Unicode became the standard in the IT world nowadays. A short overview about old charsets will be given with their advantages and disadvantages. It will be described how Unicode stores characters, which different Unicode encodings exist and which benefits Unicode brings. At last, the focus will be on how Unicode is useful in the IT world and there will be examples in which way Unicode helped in software development.

1. Introduction

What divides us humans from animals? It's the ability to speak and write in different languages and accents. In the evolution of the Homo sapiens, the ability of writing began very early. Even in the beginning of human life in the Stone Age, the Neanderthal began to draw pictures of important events on the walls of the caves. Through the further evolution process of the human race, the human began to write down every information our race knows by hand. In the middle of the 15th century, Johannes Gutenberg invented the letterpress which was a very big step in the history of text. Since then, it was possible to print texts over and over again at a very high speed. The newest big event in the history of text was the ability to display text on a digital monitor – changeable by electricity. In our modern world, the media in which text is displayed changed but the importance of these texts stayed the same as in the 15th century or before. With more than 7000 different languages all around the world, all these languages have their own characters and specialties. In the following I will explain how these characters are encoded in the computer storage, which charsets exist and why Unicode became the standard of today.

2. Characters in general

Characters are basic or atomic units of texts. Words and texts are the combination of characters.

This sequence of characters is also called a "string" and it does not need to consist of characters which follow linearly one after another. In different languages or writing systems characters can be combined in very complex ways to achieve one glyph. But even in this case texts still logically consists of a sequential order of characters. [1]

Glyphs are not necessarily consisting of one character. There are ligatures like "fi" which are one atomic unit of text but consist of two "characters". So, there are composed characters like "fi" which are used as a single character but consist of two or more different characters. [2]

There are also signs which are not used in regular texts but only in special situations or contexts and are treated like characters. Often these signs are used together with normal

characters. An example is the estimated symbol ℮, which is a special variant of the letter "e". It is not used in normal texts but in European countries on packaging to specify the conformance to specific packaging standards. This glyph is identified as character also because it is used in relation to text characters like "℮ 250g" which indicates that the weight of the product is around 250g in specific tolerance standards.
There are other logos and symbols which are not treated like characters even if they are used in context with accompanied characters or text. For example, logos consist naturally of characters like a name or abbreviation in a specific style often accompanied by an image. It would be unnatural to treat this logo like a character even if it consists of a sequence of characters. But it is possible to use a string as a replacement of a specific logo. [2]

Texts nowadays are often written and processed on computers. There are two different types of data all computer programs are built of and which a computer processes: characters and numbers. Internally characters are also stored as numbers. But the way in which the computer handles these numbers is very different to numeric values for calculations. [3]

Storing the specific appearance of a character in a numeric value would not be very efficient and useful. The better approach is to store, process and transmit only the information which character is meant. The specific visual appearance is not transmitted. So, the different systems have to interpret the meaning of the transmitted numeric value and display the meant character in the correct way. [1]
To simplify the interpretation there are different collections of characters. These collections are called charsets. Each character in a charset has a unique numeric value. But the same numeric values can appear in different charsets. To interpret a numeric represented character the exact charset in which the character was encoded is need to be known to ensure a correct decoded character. E.g., the numeric decimal value "196" is decoded by the charset "US-ASCII" as the letter "Ä" [4] but by the charset "ISO 8859-7 (Greek)" the numeric value "196" is decoded as the letter "Δ" [5].

3. History of charsets

Over history there established many different regional charsets with their own specific special characters.

One of the first charsets which were developed is the "American Standard Code for Information Interchange (ASCII)". [6] The first version of ASCII was introduced in 1964. ASCII is based on a 7-bit-byte. Each character gets represented by one byte. Characters are represented by different byte values. With the 7-bit-byte, 2^7 (= 128) characters can be referenced. 33 characters [4] are preserved for the use as control signals like start-/end-of-transmission-codes or block and record separators. 95 characters are left for representing textual characters. ASCII was developed as an American standard for English character sets and texts. The biggest disadvantage of ASCII is the fact, that only 95 characters are supported. This results in the problem that ASCII is not useable for e.g. Asian languages because these characters are not supported due to the 95 character limitation. [6]

Another very well-known charset family is the ISO/IEC 8859 family of character code standards. ISO 8859 is mainly supporting European languages. ISO 8859 codes are widely

used on several platforms, e.g. the internet. ISO 8859-1 was for a long period the default encoding on Windows. Also, on Linux and Unix, ISO 8859 is very common.
Each different sub-standard of the ISO 8859 family tries to address the needs of one or more languages. A character is represented by an 8-bit-byte. With the 8-bit-byte, 2^8 (= 256) characters can be referenced. It is not possible to represent multilingual texts with a single ISO 8859 charset encoding. [7]

The ISO 8859-1 is also called ISO Latin 1. It represents the character repertoire and the belonging character codes of the Latin alphabet No. 1. The repertoire contains the ASCII character set as well as various accented characters and special characters used in western and northern Europe languages. Each character gets represented by one 8-bit-byte. The character code numbers of the ASCII characters contained in the ISO 8859-1 is identical to the character code numbers in the ASCII character set. The additional characters which extend the ASCII repertoire occupy the code positions from 166 to 255. [7]

The ISO 8859 character codes were also adopted by Microsoft which developed their own "Windows codes". In ISO 8859 there are several code positions reserved for control characters. These characters are only barely used. In Windows codes these control characters are replaced by various printable characters, mainly punctuation marks. Due to the high market share of Microsoft the Windows codes became very important and well known. [8]

Microsoft also defined their own "Windows Latin 1" which is based on the "ISO Latin 1". In ISO Latin 1 the decimal character codes 128 to 159 are used for control characters. In "Windows Latin 1" these character codes are assigned to e.g. single quotation marks (‘), em dash (—) or en dash (–). "Windows Latin 1" became one of the most commonly used charsets in the world. In most cases when the default is said to be "ISO Latin 1" it's "Windows Latin 1". One example are web pages which are labeled to be "ISO Latin 1" encoded but contain bytes with values from 128 to 149. Browsers will automatically display them like they were encoded with "Windows Latin 1". But this cannot be implied. If there are programs which expect the "ISO Latin 1" encoding and there are bytes in the range between 128 and 159 this might cause issues. These "Windows Latin 1" specific characters may be interpreted as control characters or be ignored by the decoding program. [8]

"Windows Latin 1" is just one example. There are many other Windows codes defined.

To conclude: First charsets were developed in the United States of America. These charsets were expanded more and more by regional charsets for different languages and needs. The referencing of characters in different charsets are not unique. E.g. character 129 in "Windows Latin 1" and "ISO Latin 1" have different meanings. These results in the problem that the program which decodes the encoded text always must know the charset in which the text was encoded. If the wrong charset is provided, the decode process may result in errors or the decoding may produce a wrong text.

4. Unicode

Unicode is a charset with the purpose to solve the "charset jungle". It tries to be a universal text encoding standard which unites all the different charsets and characters in one charset

by giving every character of every language a unique identifier. Unicode's primary approach is to increase the size of the number of possible encoded characters by increasing the number of bits used to encode a character. [6] [3]

Unicode is desired to represent "plain text" in computer storage. It's not able to represent "rich text" or "styled text". [9]

a) Unicode's 10 Design principles

The Unicode Standard was designed following 10 Design principles. The principles should be handled very critically because there are conflicts between those. The standard does not specify how these conflicts should be resolved. In the following a short overview about the design principles will be given:

1. Universality
   A single set of characters is defined by Unicode which can be used in universal ways. The character repertoire must be large enough to include all characters which are likely used in texts of different languages or writing systems. Unicode cannot assume that all text is written from left to right or that all letters have uppercase and lowercase forms.
2. Efficiency
   Unicode texts are simple and efficient to process. Every character has one unique identifier with which the characters is represented internally. It's much easier to work with data in a system where the identifiers are unambiguous. Software does not have to maintain a specific state or search for special escape sequences. Character synchronization from any point in a stream of characters is quick and unambiguous. Fixed character code allows efficient sorting, searching, display and editing.
3. Characters, not glyphs
   Unicode provides a charset and assigns identifiers to characters, not to their visual appearances. There are some special cases but it is a fundamental idea of Unicode. Unicode draws a strong and important distinction between a character and a glyph. A character is a linguistic concept in written form, e.g. the Latin letter "B" or the Chinese character "水" for "water". A glyph is the concrete visual representation of a character.
   Between characters and glyphs is no one-to-one correspondence. A single character can be represented by two or more glyphs. A character may also be represented using different glyphs in different contexts. Unicode does not try to encode every possible "look" or "shape" of a character. Unicode encodes the character, the representation as glyph is not encoded. [9]
4. Semantics
   Every character has a defined meaning. Often these meanings are defined indirectly or implicitly but Unicode cares more explicit about meanings of a character than other character code standards do.

5. Plain text
   Unicode only handles plain text without any formatting, structure or text decoration. The only exception are line breaks.
6. Unification
   Unicode encodes duplicates of a character with the same identifier if they belong to the same script but are in different languages. For example, the German “ü” and the Spanish “ü” have a different meaning or pronunciation in both languages but are handled as one and the same character in Unicode.
7. Dynamic composition
   Characters with diacritic marks can be composed dynamically. Almost any character can be combined with any diacritic. For example the “̃,“ (comma with tilde) character can be dynamically composed by using the normal comma character and the combining tilde. Using dynamic composition, much more characters can be written than characters are defined in Unicode. There also can be combined multiple combining marks with a character.
8. Logical order
   The standard representation of Unicode data is in logical order in opposition to other standards which handle writing directions by changing the order of characters. The ordering principles put all diacritics after the base character to which they are added. The visual placement is not regarded. An example is the Greek capital letter omega with tonos. This characters has the tonos on the left side of the omega (Ώ) but the decomposed form of the character still consists of the omega followed by the combining tonos mark.
9. Equivalent sequences
   Many characters in Unicode are in precomposed forms like the character “é”. These characters have decompositions which are handled equivalent to the precomposed forms. Applications may treat both forms as different characters because as strings of encoded characters both variants are different. But these distinctions should not be made. The Unicode standard does not declare to prefer one of the both forms.
10. Convertibility
    Texts can be easily converted between Unicode and other character standards. This is also part of the universality principle. [10]

b) <u>Unicode character definition</u>

Most other charsets use an 8-bit word to encode characters. Unicode uses a 16-bit word to encode characters which enables Unicode to encode up to 2^{16} (= 65 536) characters without using more complicated schemes to encode furthermore characters. 65 536 characters is enough to encode the majority of characters in the majority of written languages which are in use today. [11] The newest version of Unicode (version 15.0) is able to encode up to 149 186 different characters. [12] Less common characters are represented using two 16-bit words. [6]

Unicode defines characters on a very specific way. Every character is identified by three mandatory attributes:

- the “Unicode number” which is the unique 16-bit word for every character
- a representative glyph in normal text size given to the character
- the “Unicode name” in uppercase

The “Unicode number” and “Unicode name” are unique to every character and will be never changed.
There can be some more optional attributes like:

- the “old Unicode 1.0 name” in uppercase
- other names which may be changed preceded by an equal sign
- Comments on usage of the character preceded by a bullet sign
- Cross references to other characters preceded by an arrow
- Information that specifies the character as decomposable character preceded by a ≡ or ≈ symbol [13]

The cross references often are used to warn against confusing a character with another, to present alternative names for it or to describe possible variations in the visual appearance.

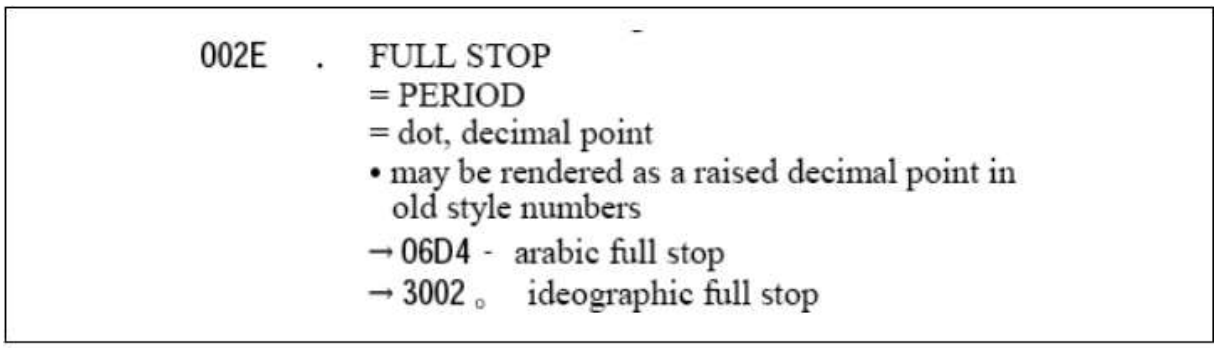

Figure 1: Unicode definition of the "full stop"-character [14]

c) Unicode encodings

The Unicode Standard defines different encodings for the Unicode charset. The first encoding is “UTF-32” (Unicode Transformation Format 32). The main principle of the “UTF-32” encoding is to map each character identifier to 4 bytes (32-bits) in storage. Unicode Identifiers are designed to use only 21 bits. This means that the “UTF-32” encoding wastes storage space of at least 11 bits per character because these are always set to zero. But there is still an advantage in using “UTF-32”. It allows a fast data access. To access a character in a string on position n, a program would just have to add 4*(n-1) to the base address of the string, because all characters are represented using the same number of bytes. [15]

The second well known Unicode encoding is “UTF-16”. “UTF-16” uses 2 byte (16-bit) to encode a character. All characters of the Unicode charset with a at maximum 16-bit long unique identifier are represented directly by 2-byte. Characters which identifier contains a number which cannot be represented by 16 bits are splitted in 2 pairs of 16 bit. One of the two 16-bit pairs contain the high value and the second 16-bit pair contain the low value of the identifier. Using “UTF-16” a character cannot be access directly because characters can be represented by 16-bit or 32-bit in storage. Due to this issue it is not possible to calculate

the correct offset to the base address of the string. “UTF-16” is as robust as “UTF-32”. If a 16-bit word is corrupted then only the referring character cannot be processed. [15]

The third Unicode encoding is “UTF-8”. “UTF-8” uses by default 8-bits to store a character. Characters of the ASCII range (0x00 up to 0x7F) can be represented efficiently. All other characters which identifier exceeds value of 0x7F are represented using 2, 3 or at maximum 4 bytes. Bytes which represent a character outside the ASCII range are identified by setting the most significant bit to 1. If a byte has the most significant bit not set it is sure to say that this byte represents an ASCII character.
“UTF-8” is relatively inefficient because it leaves many combinations unused. For a character outside the ASCII range, “UTF-8” uses two or more bytes. How characters are stored is displayed in Figure 2. The first byte identifies how many following bytes are used to represent this character. If the first byte has the most significant bit not set, there are no following bytes. If the first byte has the first 2 bits set to 1 and the third set to 0 the character is represented using 2 byte. If the first 3 bits are set to 1 and the fourth is set to 0, there are 3 bytes used to represent the character. The last combination is that the first 4 bits are set to 1 and the fifth bit is set to 0. This means that 4 bytes at total are used to represent the character. All following bytes are beginning with the highest bit set to 1 and the second highest bit set to 0.

In the following table the previous explained principle of storing character identifiers in binary using one up to four bytes is shown.

Code number in binary	Octet 1	Octet 2	Octet 3	Octet 4
00000000 0xxxxxxx	0xxxxxxx			
00000yyy yyxxxxxx	110yyyyy	10xxxxxx		
zzzzyyyy yyxxxxxx	1110zzzz	10yyyyyy	10xxxxxx	
uuuww zzzzyyyy yyxxxxxx	11110uuu	10wwzzzz	10yyyyyy	10xxxxxx

Figure 2: Storing a character in binary using variable number of bytes [16]

Interpreting UTF-8 encoded data is very simple. If the first bit of a byte which contains encoded UTF-8 data is 0, the byte is a single octet and represents a Basic Latin character. Otherwise the second bit is of interest. If the first bit is set to 1 and the second bit is 0, the byte is a second, third or fourth byte of a multibyte representation of a character. If the first bit is set to 1 and the following bits are 10, 110 or 111 it tells that 2, 3 or 4 bytes are used to represent a character.
Decoding UTF-8 data is also very simple. One byte is taken and compared to the “Octet 1” column of the table above. If the byte has following bytes these are also read. Then the binary code number is constructed using the read bit sequences. This operations can be implemented efficiently as operations on bit fields. If data does not match any of the predefined patterns an error is signaled.
Like in UTF-16, in UTF-8 the n-th character of a string cannot be accessed directly because a

character is not represented by a fixed number of bytes. If a byte in a character representation multibyte is corrupted other characters will still be processed correctly. [15]

d) Criticism

Unicode does not only have supporters. There is also criticism on the character standard. The most criticism is referred to the complexity of Unicode. Unicode has to be very complex because languages itself are a very complex topic. The basic concept and principles of Unicode are easy to understand, but in detail Unicode contains many different concepts, definitions and algorithms. Many old character codes are much simpler than Unicode because they ignored different languages, writing systems and notation systems. Unicode is very universal. But with universality also comes complexity. Most of the other criticism also refers to the complexity aspect. [17]

Another critique aspect is that Unicode is inefficient. Unicode would use two bytes for each character. This ends up in doubling the size of a text file which is encoded in Unicode in opposite to the same text file encoded in a charset which encodes every character by only 8-bit. But this statement is only partially true. As explained before, Unicode has several different encoding forms. Using UTF-8, the size of a text file remains exactly the same then if ASCII would have been used. UTF-8 uses only 8-bit for one character. This only works if the text consists of ASCII characters only. If any other characters are used these other characters may be encoded using 16-bit, 24-bit or 32-bit. So at least when using the UTF-8 encoding, a text may be bigger than in ASCII but it may also be the same size. This depends on the language and the characters which are used. In Figure 3 the number of bytes used to store a character is shown for the different Unicode encodings. It can be seen that the used storage depends on the used characters and the chosen encoding. [17]

Class of characters	Range of characters	UTF-8	UTF-16	UTF-32
Basic Latin (ASCII)	U+0000 to U+007F	1	2	4
Latin 1 Suppl., …, Thaana	U+0080 to U+07FF	2	2	4
Rest of BMP	U+0800 to U+FFFF	3	2	4
Outside BMP	U+10000 to U+10FFFF	4	4	4

Figure 3: Bytes used to store characters in different UTF encodings [18]

In combination to the inefficiency critique, the critique on the reasonability of supporting 100 000 characters came up. The character repertoire of Unicode is very big. Most applications and use cases only use a very small amount of the character repertoire. So, the claim started not a subcode exists which only contains for example 1000 of the usually used characters instead of using the Unicode standard which supports all 100 000 characters. The answer is simple: A Software which supports Unicode must be able to process all Unicode characters not only a subset of 1000. It needs to contain a font which can display all 100 000 characters. If a subset would be created, the whole purpose of Unicode, the unification of all charsets, would be useless. But the Unicode standard contains a paragraph which says, that

applications are allowed to be ignorant to characters they can't process as long as these characters are not deleted or changed. [17]

5. Unicodes influence on the IT world

Before Unicode existed, multilingual applications were a major issue. The problem of representing characters in multiple languages existed. An example is software which is executed on computers in Russian language and in German language. Here the ability is given to design the software in that way that every language has their own tool descriptions which does not lead into an issue regarding the character encoding. The issue starts when a document should be created by the software which should contain Russian and German text in one document. It is needed to be guaranteed that Latin and Cyrillic letters can be used inside the same document. This problem was finally solved when Unicode was introduced. [1]

Unicode is not the solution to all problems in developing international multilingual software. Unicode alone is not able to produce internationalized software at its own but it solves one important and particular problem in writing internationalized software: text needs to be displayed in different languages on different computer systems without getting tripped up in dealing with multiple encoding standards for different languages. [9] Texts can now also mix different languages and characters inside of one text document. Character identifiers are now no longer ambiguous as they were before using multiple different charsets with the same identifier values for different characters. In Unicode every character has an own unique identifier and no additional information is needed to interpret the given identifier value correctly. [6]

Unicode had a very big impact on the IT world and many applications support Unicode. For example widely used software like Microsoft Windows, Mac OS X and Linux has supported Unicode for years already. Although these operating systems know and support Unicode there are still applications running on these operating systems which does not support Unicode. Often the display or printing of Unicode characters fail because fonts are still incomplete in covering the total amount of characters which can be represented using Unicode. This is changing but still a problem which developers need to be aware of. [3]

Unicode also made conversion between charsets way easier. Unicode is a superset of all other used character encodings. This includes the advantage of helping on conversion between encodings. Character data of any character code can be converted into Unicode without losing information. If it is converted back, it is the exactly same data than it was before the conversion. If a system needs to be provided which can convert text between different encodings, the number of needed converters is way smaller. If n different encodings are supported only 2n converters are needed (not n^2 converters as without Unicode). [6]

Unicode is very flexible regarding the area of use. For example for the use on the internet. Even if the internet connection speed of many households are fast nowadays, it is still important to focus on efficiency of data. Storing large files on a server is not a problem anymore due to the fact that hard disks are very cheap today. But the transmission time for

transmitting a file over the network is proportional to the file size. For the use on the internet, Unicode encoding UTF-8 is recommended because it has the lowest data overhead for characters because it uses 8-bit for ASCII and extends up to 32-bit for special characters. UTF-8 is explicitly recommended by the Internet Engineering Task Force (IETF) to be used for all text on the Internet. For data processing and programming, UTF-16 or UTF-32 are more recommended. UTF-32 allows to access the nth character of a file directly by adding an offset to the starting address because all characters are stored by fixed 32-bit length which makes it very useable in working with textual data. UTF-16 is internally used by all Windows versions today. This makes it very efficient to use UTF-16 for programs which use the built-in functions of Windows. [19]

Even if there are many benefits and positive influences Unicode had on the IT world, there were and are still some problems with it.
One example are data types in tables in a MySQL database. There the encoding “utf8” exists which programmers may assume that they can use Unicode UTF-8 encoded text with this encoding in the database. In fact, this is not true. The “utf8” encoding only supports 3 bytes per character. But the Unicode UTF-8 encoding can contain up to 4 bytes per character. This problem is fixed nowadays by introducing the “utf-mb4” (utf-8 multibyte 4) encoding which finally supports Unicode UTF-8 encoding with up to 4 bytes per character.
Another example is PHP. Many web applications are written using PHP. PHP supports only 8 bit characters by standard. The module “mb_string” needs to be activated to allow working with Unicode characters. This module is included since PHP 4.0.6. Applications which are programmed on older versions are not compatible to Unicode.
At general there is a problem in interapplication communication if one application supports and uses Unicode and the other doesn’t. If applications which don’t support Unicode gets a input with 4 byte Unicode Characters the applications often crash. One simple example are WIFI routers. Many WIFI routers support Unicode. The WIFI name can contain Unicode data. Some older “smart TVs” don’t support Unicode. Every time the “smart TV” tries to connect to the WIFI it crashes. [20]

6. Conclusion

Unicode revolutionized the way how characters were used in the IT world. It solved the problem of having many different charsets which only supported a small amount of characters for a very specific language or region. It also solved the problem of ambiguous identifiers between the different charsets. Unicode provides one charset which unites all characters which are used. Additionally, it provides different encodings which have different focus on efficiency or usability. Unicode is the standard for encoding texts in the IT world since it brought big advantages in multilingual software development, conversions between texts and flexibility between different use cases. But there are still problems between applications which support Unicode and applications which don’t. It cannot be adopted that every application can process Unicode data. This is one of the biggest issues today. In future this problem will disappear because nearly all new devices use Unicode and the older devices which don’t use Unicode will break and be replaced with new ones.

Reference list

[1] J. K. Korpela, „Unicode Explained,“ O'Reilly Media, Inc., 2006, pp. 6-10.

[2] J. K. Korpela, „Unicode Explained,“ O'Reilly Media, Inc., 2006, pp. 11-15.

[3] J. K. Korpela, „Unicode Explained,“ O'Reilly Media, Inc., 2006, pp. 3-5.

[4] „Charset.org,“ [Online]. Available: https://www.charset.org/charsets/us-ascii. [Zugriff am 18 12 2022].

[5] „Charset.org,“ [Online]. Available: https://www.charset.org/charsets/iso-8859-7. [Zugriff am 18 12 2022].

[6] R. Gillam, „Unicode Demystified,“ Addison-Wesley Professional, 2003, p. Chapter: What Unicode Is.

[7] J. K. Korpela, „Unicode Explained,“ O'Reilly Media, Inc., 2006, pp. 124-126.

[8] J. K. Korpela, „Unicode Explained,“ O'Reilly Media, Inc., 2006, pp. 127-129.

[9] R. Gillam, „Unicode Demystified,“ Addison-Wesley Professional, 2003, p. Chapter: What Unicode Isn't.

[10] J. K. Korpela, „Unicode Explained,“ O'Reilly Media, Inc., 2006, pp. 157-161.

[11] „unicode.org,“ [Online]. Available: https://www.unicode.org/Public/UCD/latest/charts/CodeCharts.pdf. [Zugriff am 18 12 2022].

[12] „Unicode Standard 15.0.0,“ 13 09 2022. [Online]. Available: https://www.unicode.org/versions/Unicode15.0.0/. [Zugriff am 18 12 2022].

[13] J. K. Korpela, „Unicode Explained,“ O'Reilly Media, Inc., 2006, pp. 16-20.

[14] J. K. Korpela, „Unicode Explained,“ O'Reilly Media, Inc., 2006, pp. 21, Figure 1-2.

[15] J. K. Korpela, „Unicode Explained,“ O'Reilly Media, Inc., 2006, pp. 301-311.

[16] J. K. Korpela, „Unicode Explained,“ O'Reilly Media, Inc., 2006, pp. 306, Table 6-1.

[17] J. K. Korpela, „Unicode Explained,“ O'Reilly Media, Inc., 2006, pp. 203-207.

[18] J. K. Korpela, „Unicode Explained,“ O'Reilly Media, Inc., 2006, pp. 327, Table 6-5.

[19] J. K. Korpela, „Unicode Explained,“ O'Reilly Media, Inc., 2006, pp. 326-329.

[20] MacLemon, „Emoji, wie funktionieren die eigentlich?,“ in *GPN19*, 2019.

Figures: