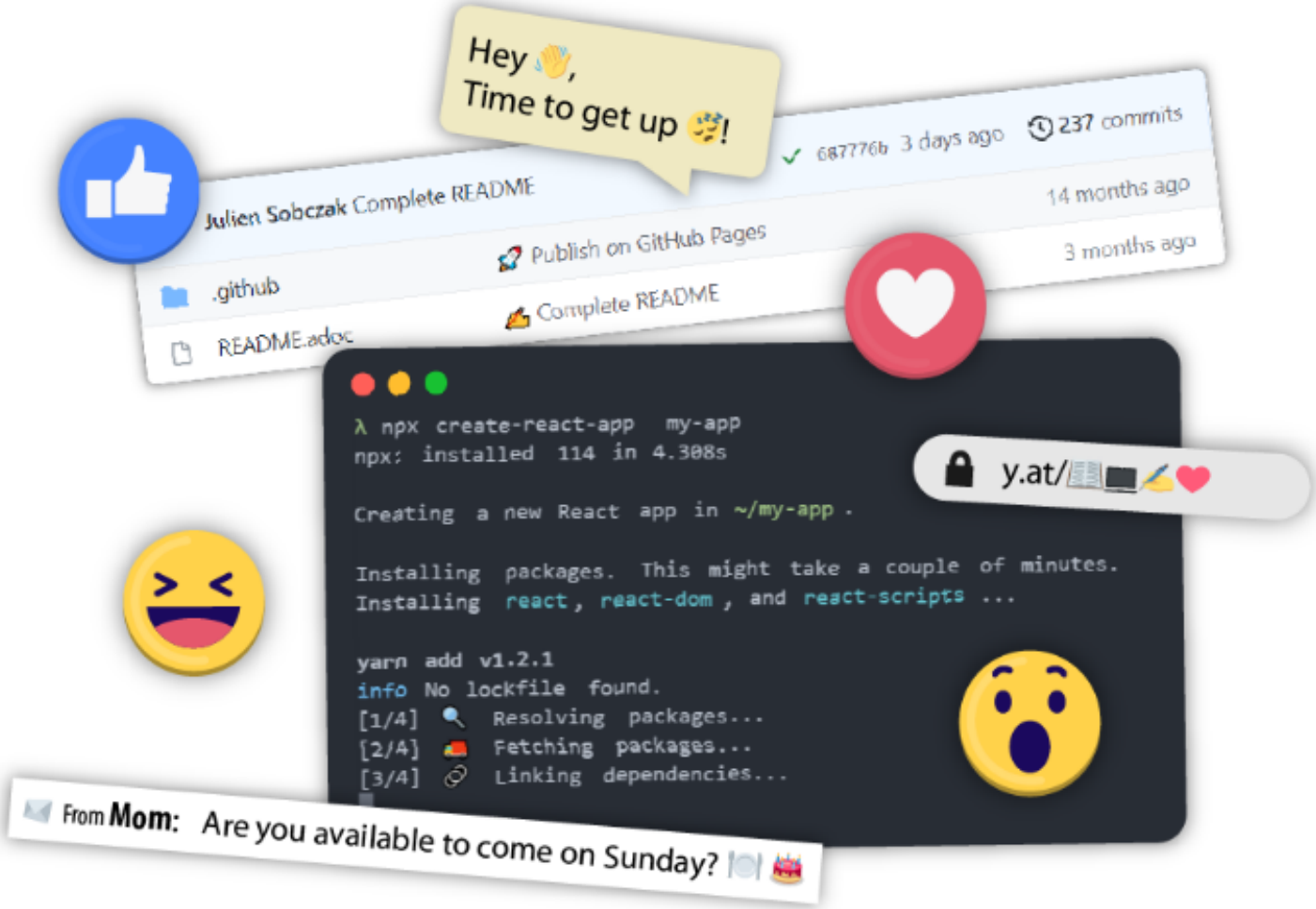
Unicode Re-explained for curious developers

> *Our goal is to make sure that all of the text on computers for every language in the world is represented but we get a lot more attention for emojis than for the fact that you can type Chinese on your phone and have it work with another phone.*
>
> — *Unicode Consortium co-founder and president Mark Davis*

Unicode is what allows me to say hello in over 100 languages:

(Amharic), 你好 (Chinese), γεια (Greek),) שלום
Hebrew), こんにちは (Japanese),) سلام
Persian), cześć (Polish), Привет (Russian), and Unicode is what allows you to understand the difference between "You are funny 🤣" and "You are funny 👎."

The web would not be the same without Unicode. But Unicode is not magic, and developers have to understand it to create applications running in a multilingual world.



The Story

- Why Unicode makes the web possible.
- Why emojis exist in Unicode.
- How you can suggest a new emoji.

The Standard

- Why Unicode is not just a character set.
- How characters are allocated in the codespace.
- Why UTF-8 is different from Unicode, and why Unicode is different from UTF-8.

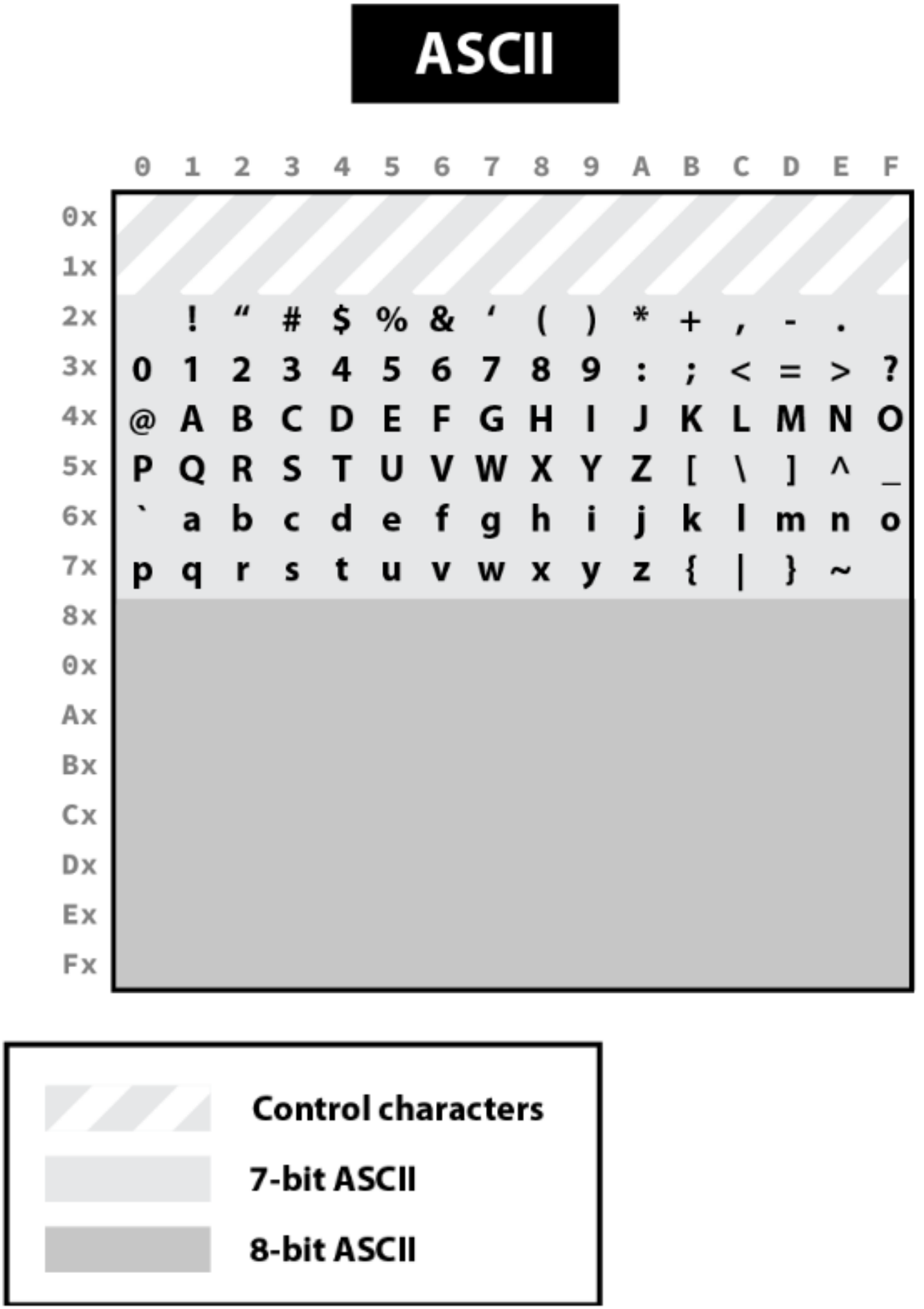
The Implementation

- Why **"à" == "à"** is false in most languages.
- Why the **pistol emoji** is now a toy.
- **How emojis are implemented** on your device.

The story of writing begins with cave paintings, but the first true writing systems were the Egyptian hieroglyphs. Hieroglyphs were lost during the medieval period, but they inspired the first alphabets, including the Latin alphabet.

Computers store letters and other characters by assigning a number to each of them, a process called character encoding. The earlier character encodings were limited and did not cover characters for all the world's languages.

ASCII was the first widely used encoding and used only seven bytes, enough to represent 128 unique characters. This was sufficient for English, but not for other languages with more characters.

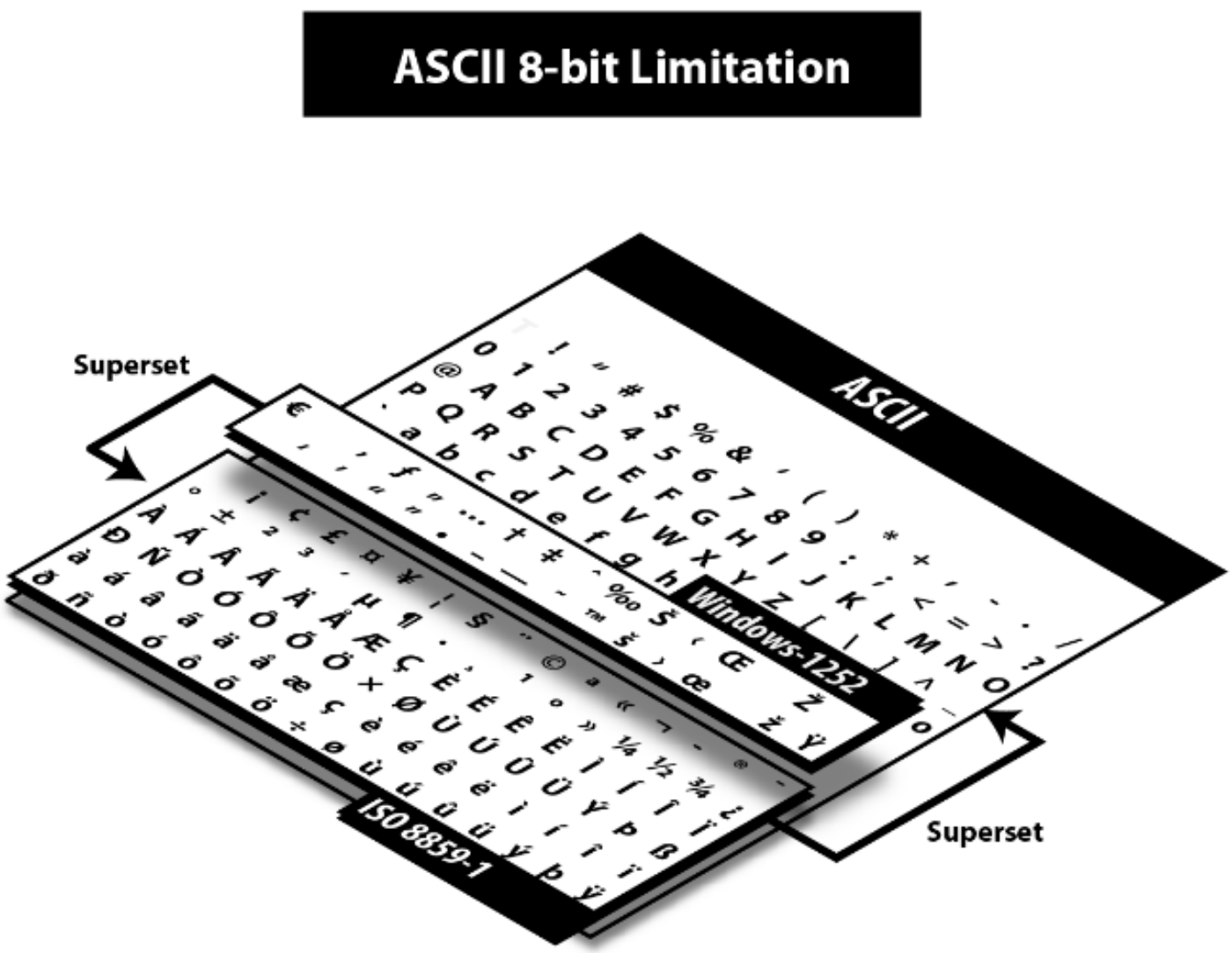


Note that ASCII reserves the first 32 codes for non-printable control characters, such as line feed and end of file.

People wanted more characters, so new character encodings were created that used the other 128 bytes available. The most famous of these were ISO 8859-1 and Windows-1252, with the latter being a superset of the former.

These encodings supported more characters than ASCII, but they were still limited. They did not support all the characters needed for all the world's languages.

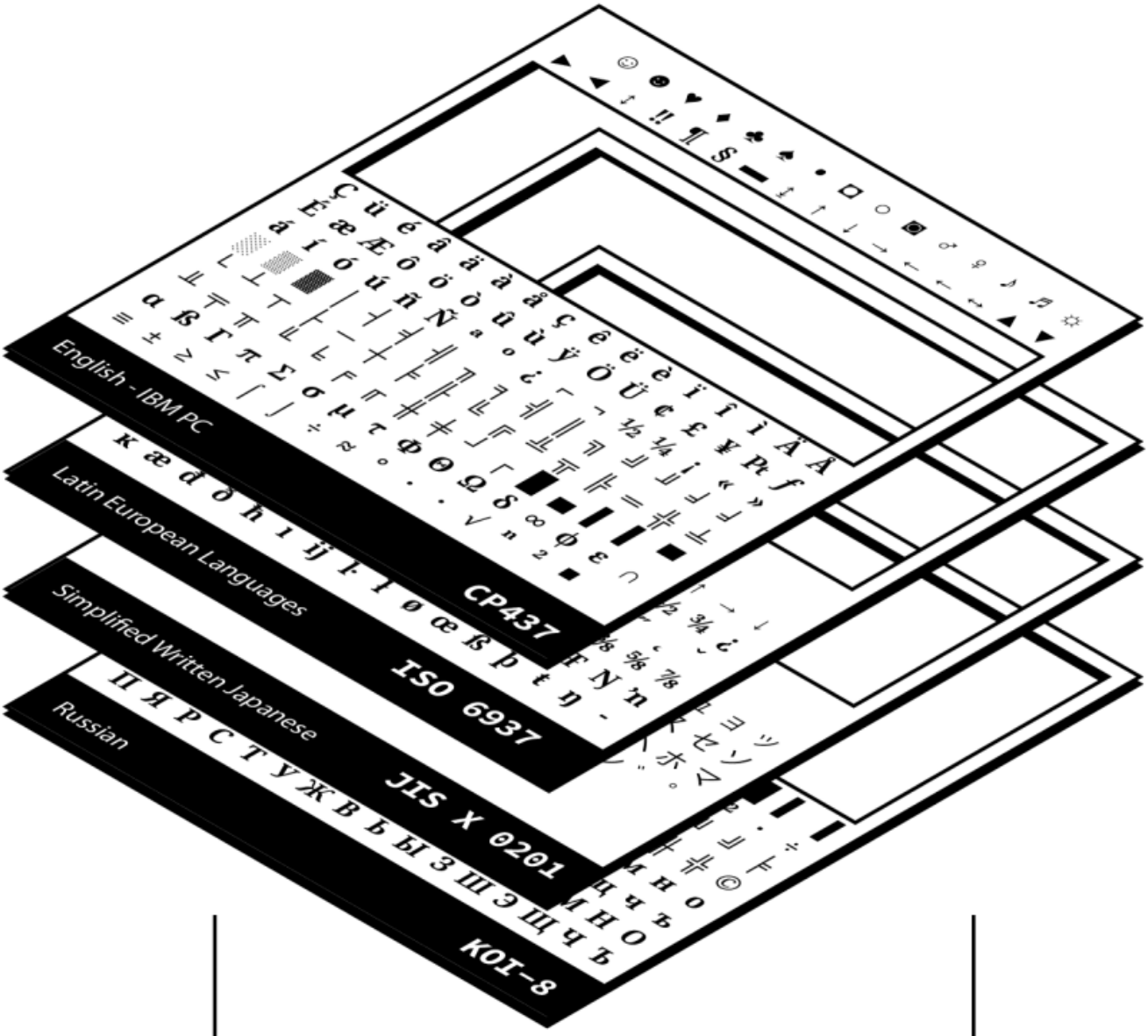
This led to the development of Unicode, a universal character encoding standard that includes characters from all major languages and writing systems. Unicode is a 16-bit system, which means that it can represent up to 65,536 characters.

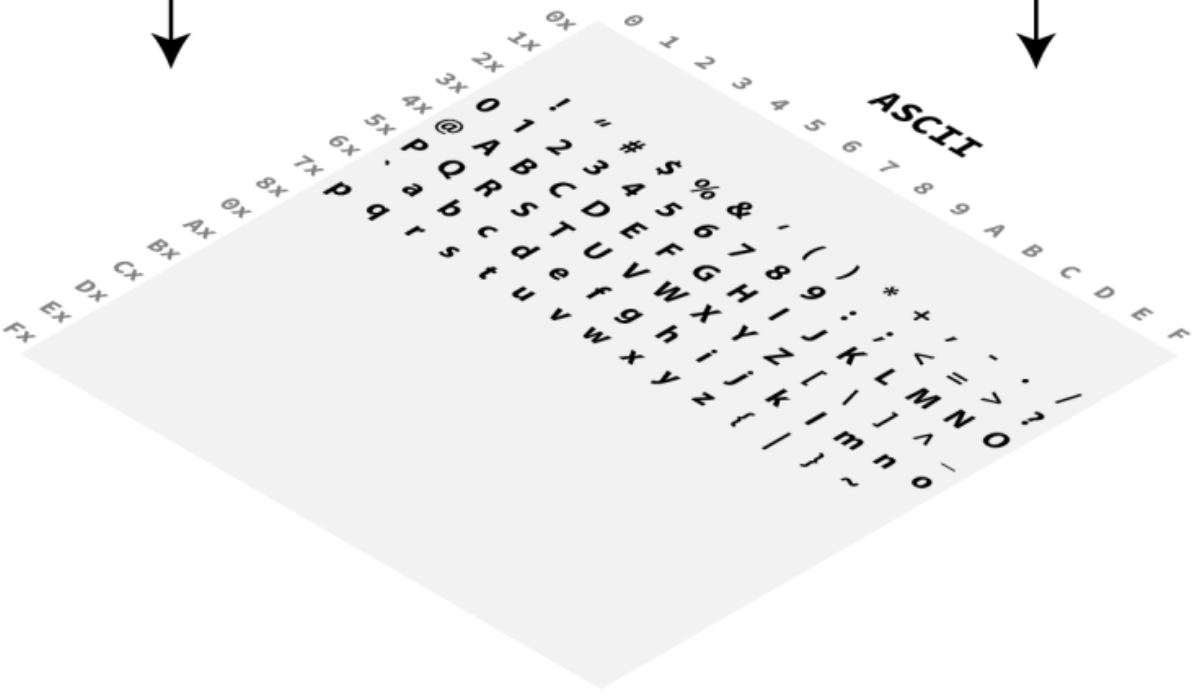


Common character sets extending ASCII

These encodings are just two examples among so many others.

Extended ASCII Character Sets





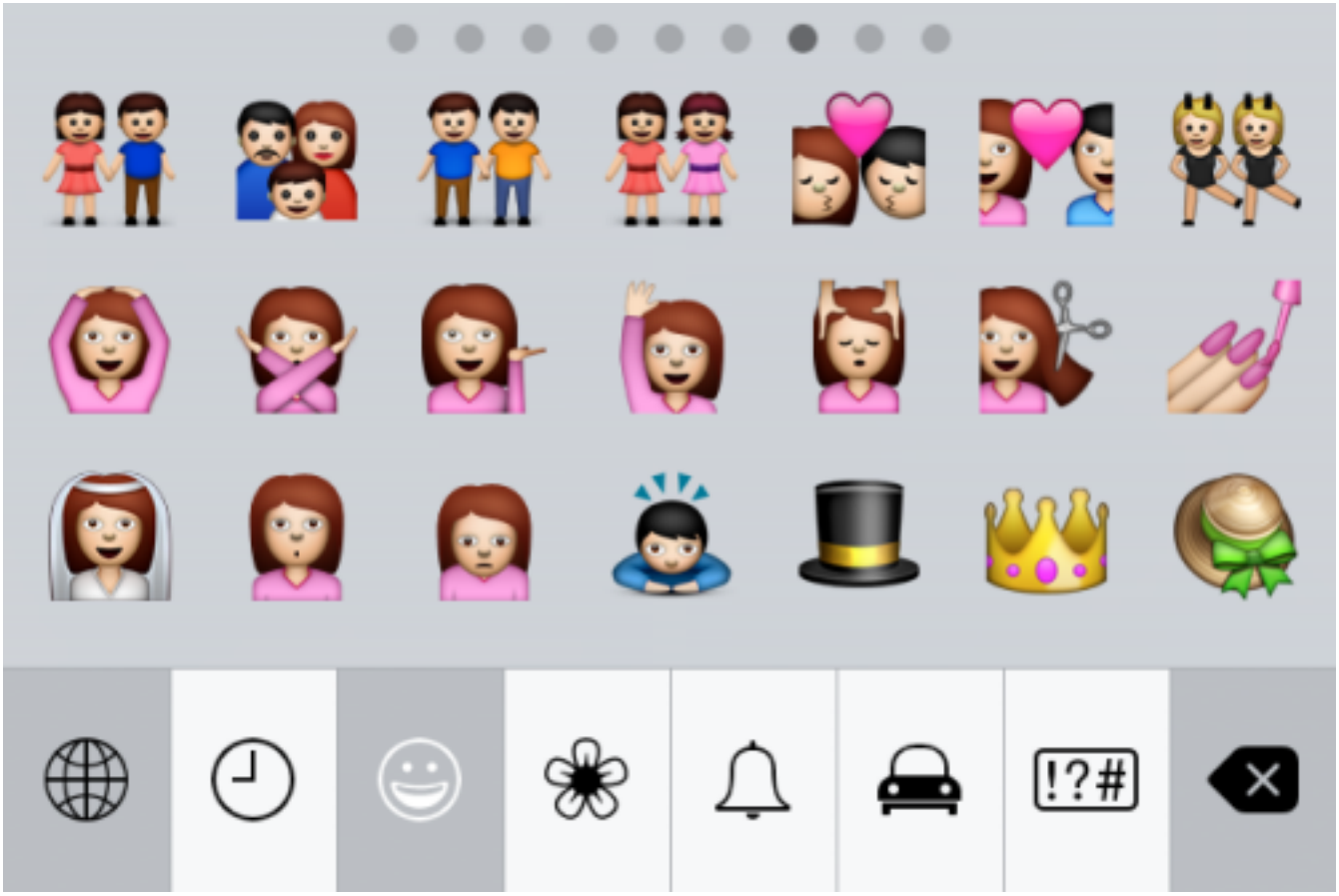
In practice, whenever textual data was exchanged between different programs or computers, the risk of corruption was high as different character encodings often use different codes for the same character.

Moreover, computers did not support all character encodings, and many languages lacked character support altogether. Clearly, 256 unique codes were nowhere near enough, especially to create multilingual documents.

THE RISE OF UNICODE

The origins of Unicode dates back to 1987, but the name Unicode first appeared the next year in the document Unicode 88. The intent of Unicode was to create "a unique, unified, universal encoding," so that computers would only have to implement a single encoding to support all languages.

Unicode began with a 16-bit design (i.e., 65,536 codes "to encompass the characters of all the world’s living languages"[3]), but was extended in 1996 to support more than a million code points.



The motivation was to allow the encoding of many historic scripts

(e.g., the Egyptian hieroglyphs) and thousands of rarely used or obsolete characters that had not been anticipated as needing encoding

(e.g., rarely used Kanji or Chinese characters, many of which are part of personal and place names, making them rarely used, but much more essential than envisioned in the original architecture of Unicode). History is always surprising.

In the meantime, the Unicode Consortium was created in 1991. This nonprofit organization, which counts only three employees, still has the same ambitious goal of replacing all existing character encodings.

This goal has almost become a reality. More than 95% of the Internet uses Unicode (it was just 50% a decade ago), and almost all electronic devices support Unicode too.

This organization is funded by membership fees (from $75 for an individual to $21,000 for a full-member company) and donations, but you can also support them by adopting a character for $100 (or $1000-$5000 for exclusivity).

THE RISE OF EMOJIS

The number of available code points exploded when Unicode dropped the 16-bit limitation.

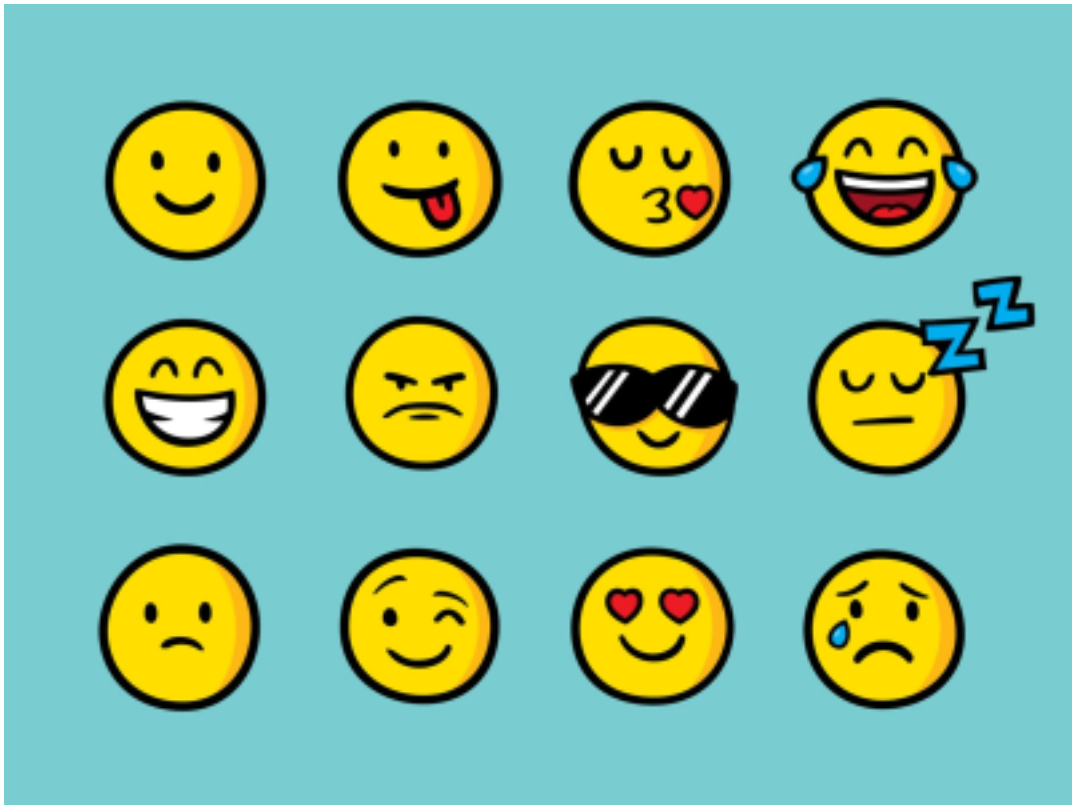
If 65,536 codes may seem a lot at that time, Unicode was now able to represent more than one million codes! Not all codes are in use. Unicode is an evolving standard.

The current version Unicode 13.0.0 uses "only" 143,859 codes, including more than 2000 special characters that we call emojis.

Emojis didn’t appear with Unicode.

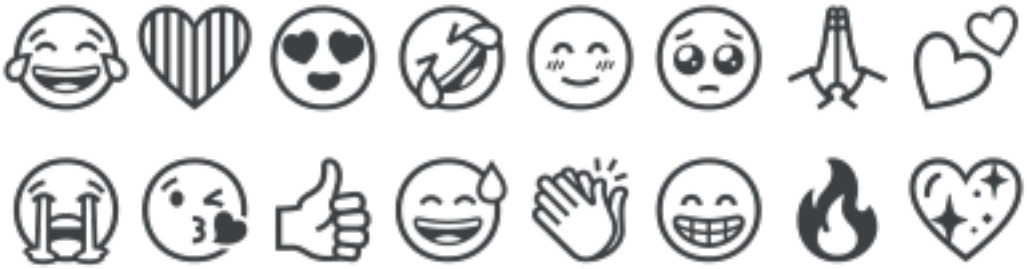
They were born in the 90s in Japan (絵文字 [emodʑi] means "picture character"), and must not be confused with emoticons such as :-), which are text-based.

Emojis were called **smileys** at that time. They were used mainly by mobile manufacturers and implemented using custom fonts like Wingdings:



The first emojis were created in 1999 by Shigetaka Kurita, a Japanese engineer at NTT DoCoMo. Kurita wanted to create a way for people to communicate more easily using their mobile phones, and he came up with the idea of using small pictures to represent emotions and ideas.

The first emojis were simple black-and-white images, but they quickly became popular in Japan. In 2007, Apple released the iPhone, which included a built-in emoji keyboard. This helped to make emojis even more popular around the world.



Unicode added support for emojis in 2010. This means that emojis can now be used on all major platforms, including computers, smartphones, and tablets.

Today, there are over 3,600 different emojis available. They are used by people all over the world to communicate with each other in a more fun and expressive way.

Emojis are a great way to add personality and emotion to your online communication. They can also be used to make your messages more visual and engaging.

If the receiver of your message didn’t have the font on his device, letters were displayed instead. For example, the national park pictogram  was available in Webdings at 0x50, which corresponded to the capital letter P encoded in ASCII.

To solve this problem, new character encodings were introduced to not mix characters and emojis by using different codes. Unicode was in danger.

Therefore, Google employees requested that Unicode looks into the possibility of a uniform emoji set. As a result, 722 emojis were released in 2010 as Unicode 6.0, and each new version now integrates new emojis. Unicode entered a new era.

Emoji standardization has put pressure on the Unicode Consortium, overtaking the initial focus on standardizing characters used for minority languages.

But the desire for more emojis has put pressure on vendors too to improve their Unicode support, leading to better support for Unicode’s minority languages.

A good example of a win-win situation. Emojis contribute to preserving the writing of the past while making the writing of the future more fun.

The primary function of emojis was to fill in emotional cues otherwise missing from typed conversations (😢🤣😉😍🥳). But emojis have been extended to include a lot more (👶👰🧛🕶🐼🍀🍘🍩⚽🚴✈).

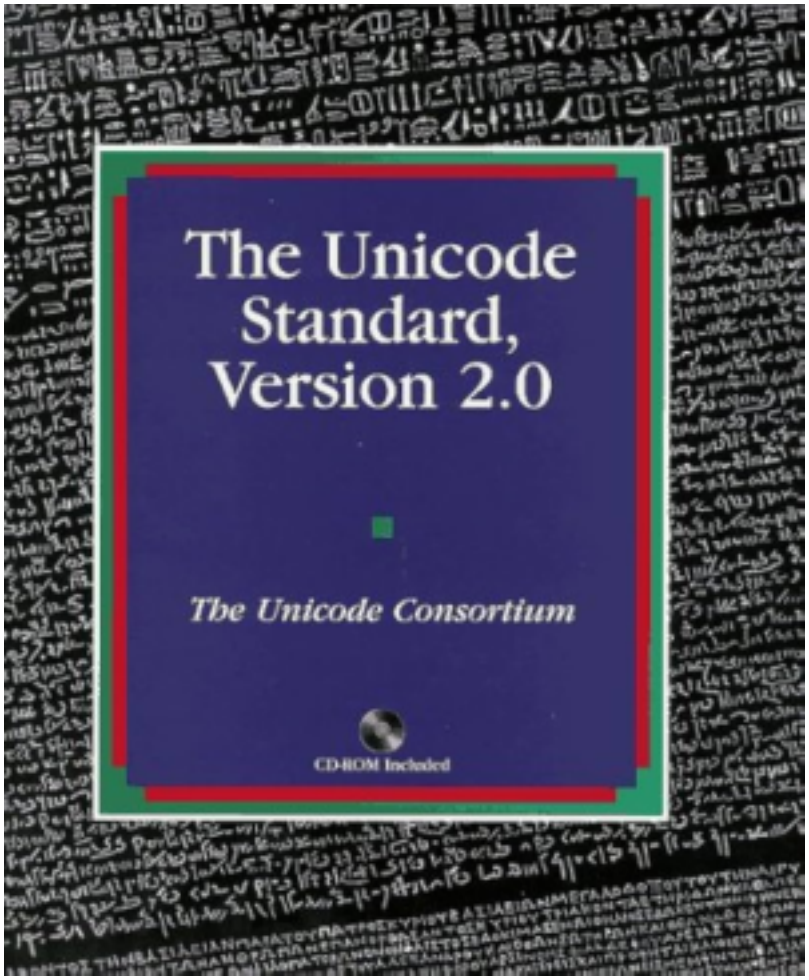
Now, when the Unicode Technical Committee meets quarterly to decide which new characters will be encoded, they also decide about new emojis.

Any individual or organization can suggest a new emoji by writing a proposal (the proposal for a troll emoji is a 10-page document using Google Trends, and film references to justify its introduction). The selection process uses well-documented rules.

A quorum of half of the Consortium’s full members is required. There are currently ten full members, only one of which, the Ministry of Endowments and Religious Affairs of Oman, is not a tech company. The other nine are Adobe, Apple, Facebook, Google, IBM, Microsoft, Netflix, SAP, Salesforce, and a newcomer, Yat.[4]

THE STANDARD

The **Unicode Standard** is a reference document that defines the Unicode character encoding. It is freely accessible online.



The Unicode Standard defines a **numeric value (code point)** and a name for every defined character. It also associates a rich set of semantics with each encoded character, such as case and directionality.

The **Unicode Character Table** has an overall capacity for more than 1 million characters, which is more than sufficient for all the world's languages, as well as currency symbols, punctuation marks, mathematical symbols, technical symbols, geometric shapes, dingbats, and emojis.

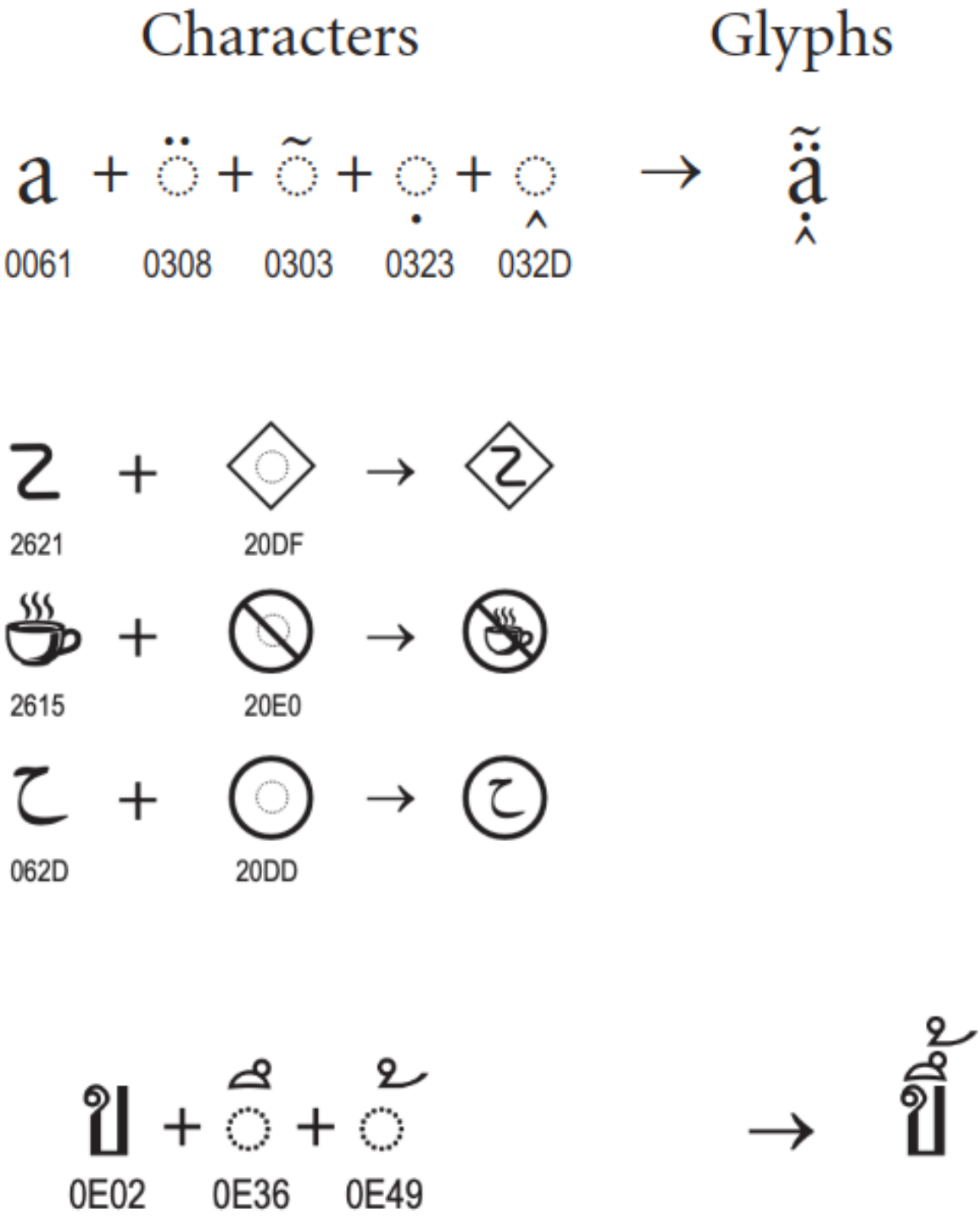
A character is an abstract entity, such as **"latin capital letter a"** or **"bengali digit five,"** defined by a unique code point. The representation of a character on screen, called a glyph, depends on the software implementation and the fonts available on the device.

Unicode includes compatibility characters, which are characters that would not be included if they were not present in other standards. The idea is to make it easy to have a **one-to-one mapping between the Unicode Standard and other standards.**

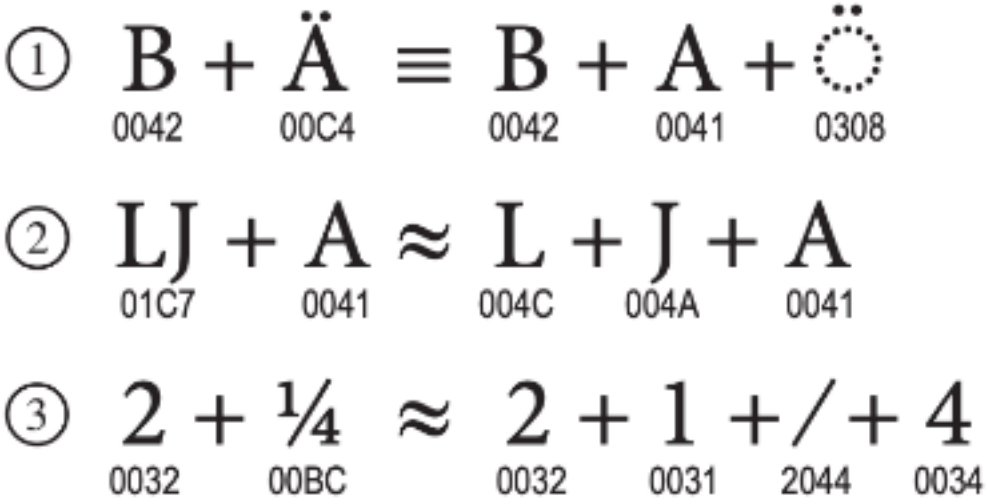
hakatashi/**unicode-map**

The Unicode Map Project

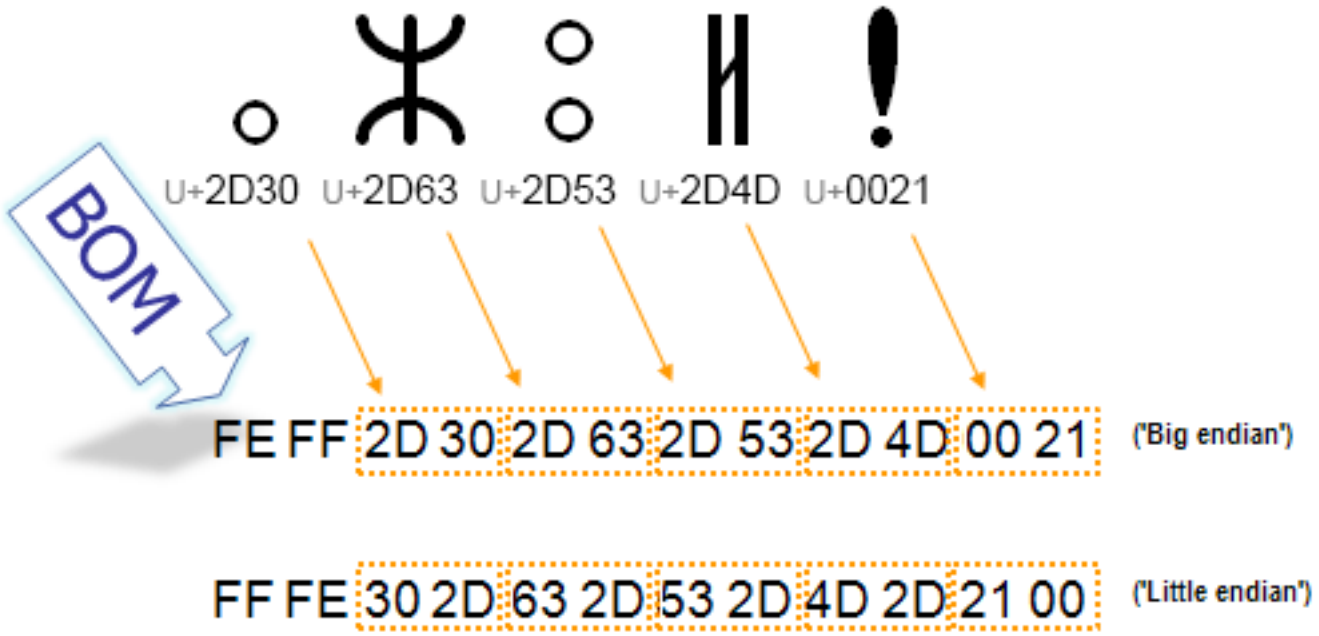
Unicode also includes **combining characters,** which are characters to be positioned relative to the previous base characters. Accents are implemented using combining characters, so U+0065 e + U+0302 ◌̂ → ê, and symbols can be composed using the same mechanism.



Unicode recognizes ***equivalent sequences***, which are two sequences using different Unicode codes to represent the same character.



Unicode includes special characters. For example, UTF-16 and UTF-32 are sensible to the endianness of the hardware and often include a **Byte Order Mark (BOM)**, which is composed of two code points—U+FEFF zero width no-break space and U+FFFE (a special character).



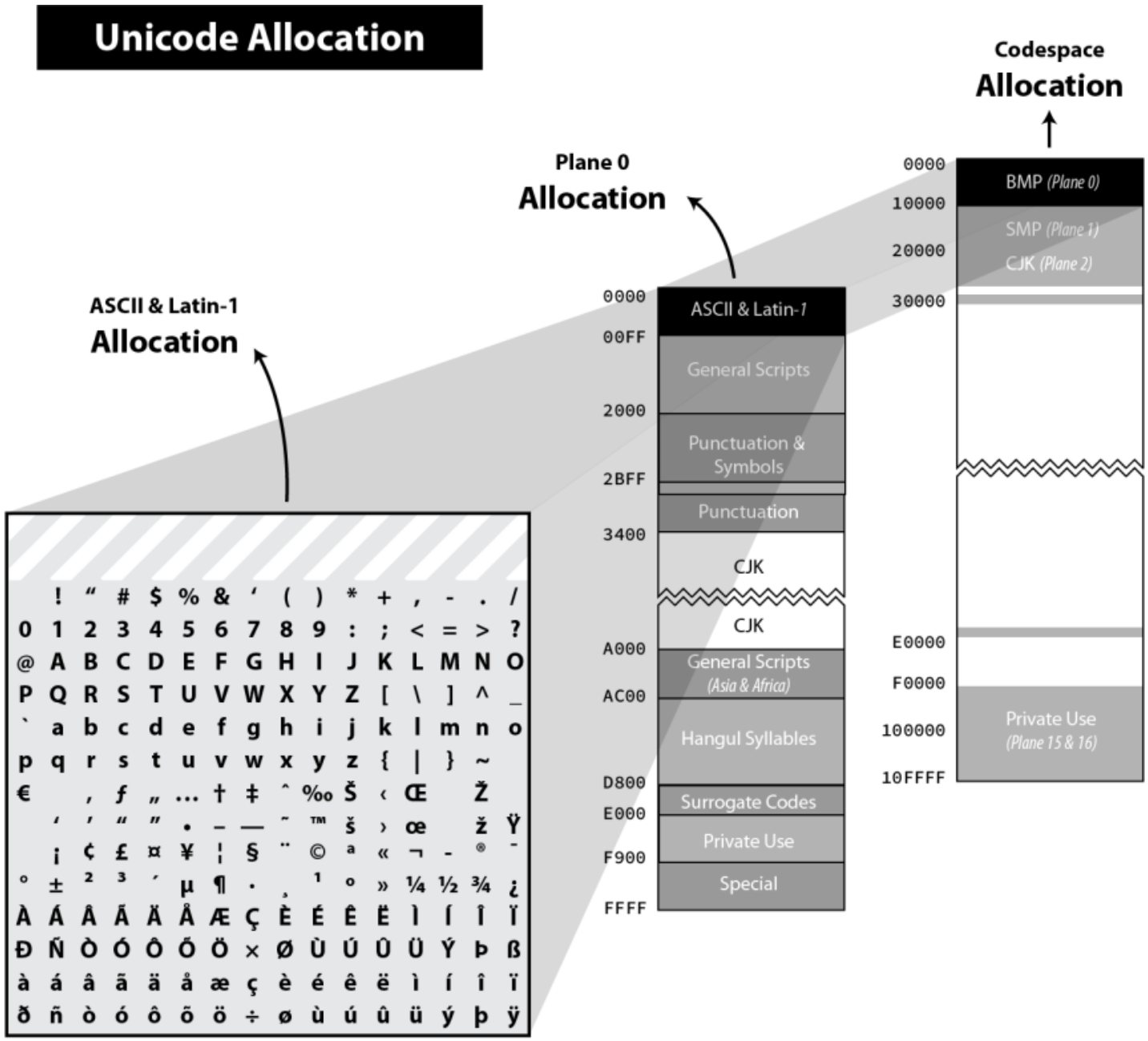
Implementations can detect which byte ordering is used in a file based on the order of these two code points. Another common example is the U+FFFD **replacement character** used to represent “unknown” characters.

We will talk more about these particularities when covering the implementation.

```
71 2.1.1  1 byte  (U-00000000):      "␀"
72 2.1.2  2 bytes (U-00000080):      "PAD"
73 2.1.3  3 bytes (U-00000800):      "□"
74 2.1.4  4 bytes (U-00010000):      "□"
75 2.1.5  5 bytes (U-00200000):      "耀□"
76 2.1.6  6 bytes (U-04000000):      "□□□"
77
78 2.2  Last possible sequence of a certain length
79
80 2.2.1  1 byte  (U-0000007F):      "□"
81 2.2.2  2 bytes (U-000007FF):      "□"
82 2.2.3  3 bytes (U-0000FFFF):      "□"
83 2.2.4  4 bytes (U-001FFFFF):      "□"
84 2.2.5  5 bytes (U-03FFFFFF):      "□□"
85 2.2.6  6 bytes (U-7FFFFFFF):      "□□□"
86
```

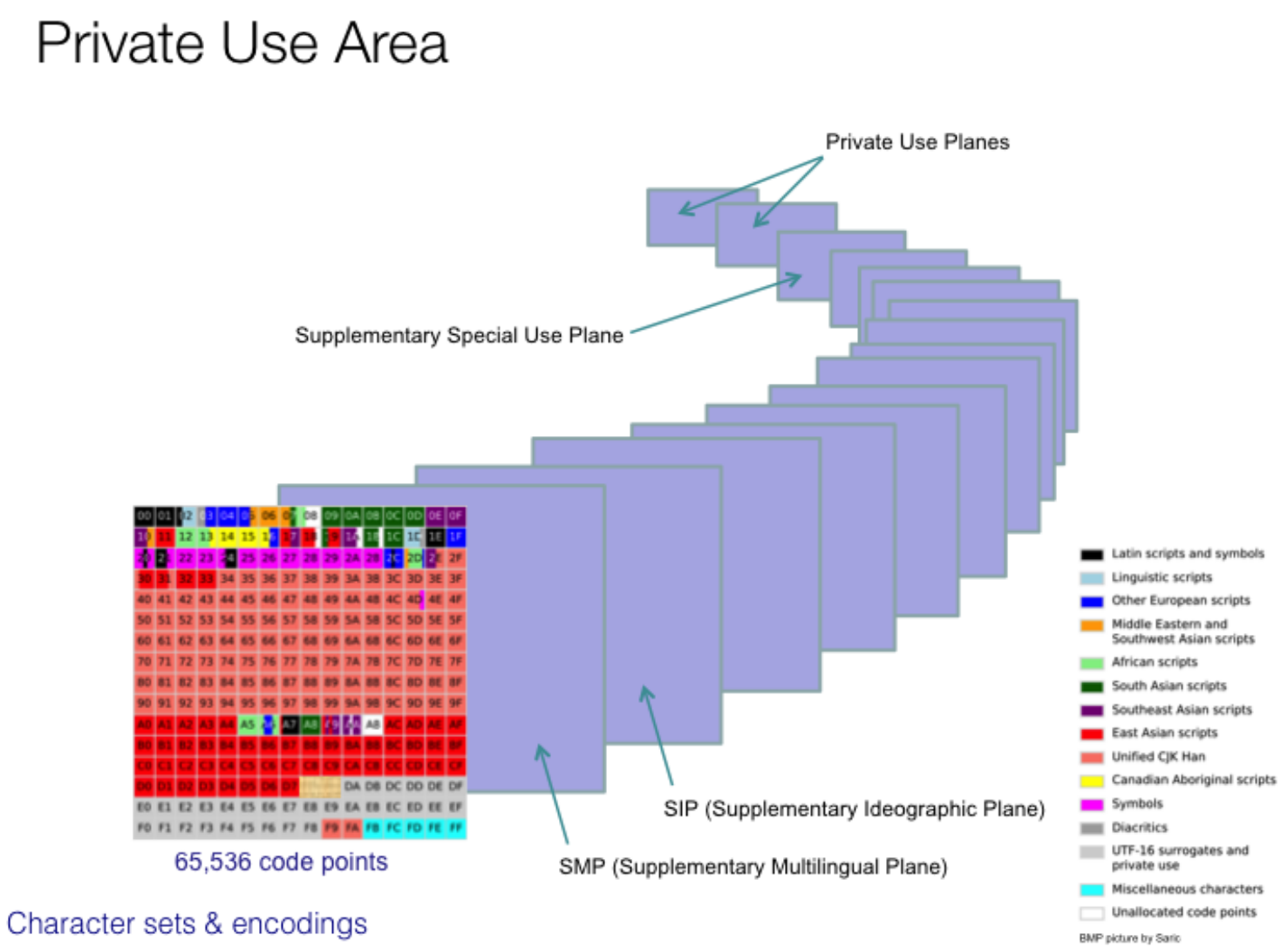
THE UNICODE CHARACTER TABLE

The **Unicode Character Table** is a table that lists all the characters in the Unicode standard. The characters are organized into planes, which are further divided into blocks.



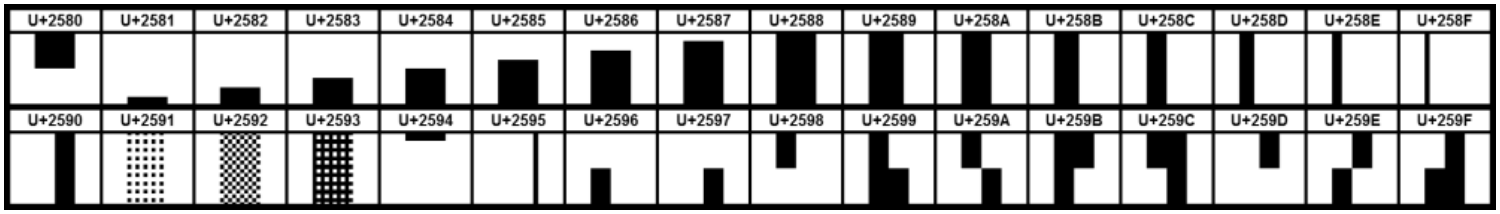
Planes

A plane is a group of 64,000 code points. There are 17 planes in the Unicode standard, but not all of them are currently in use.



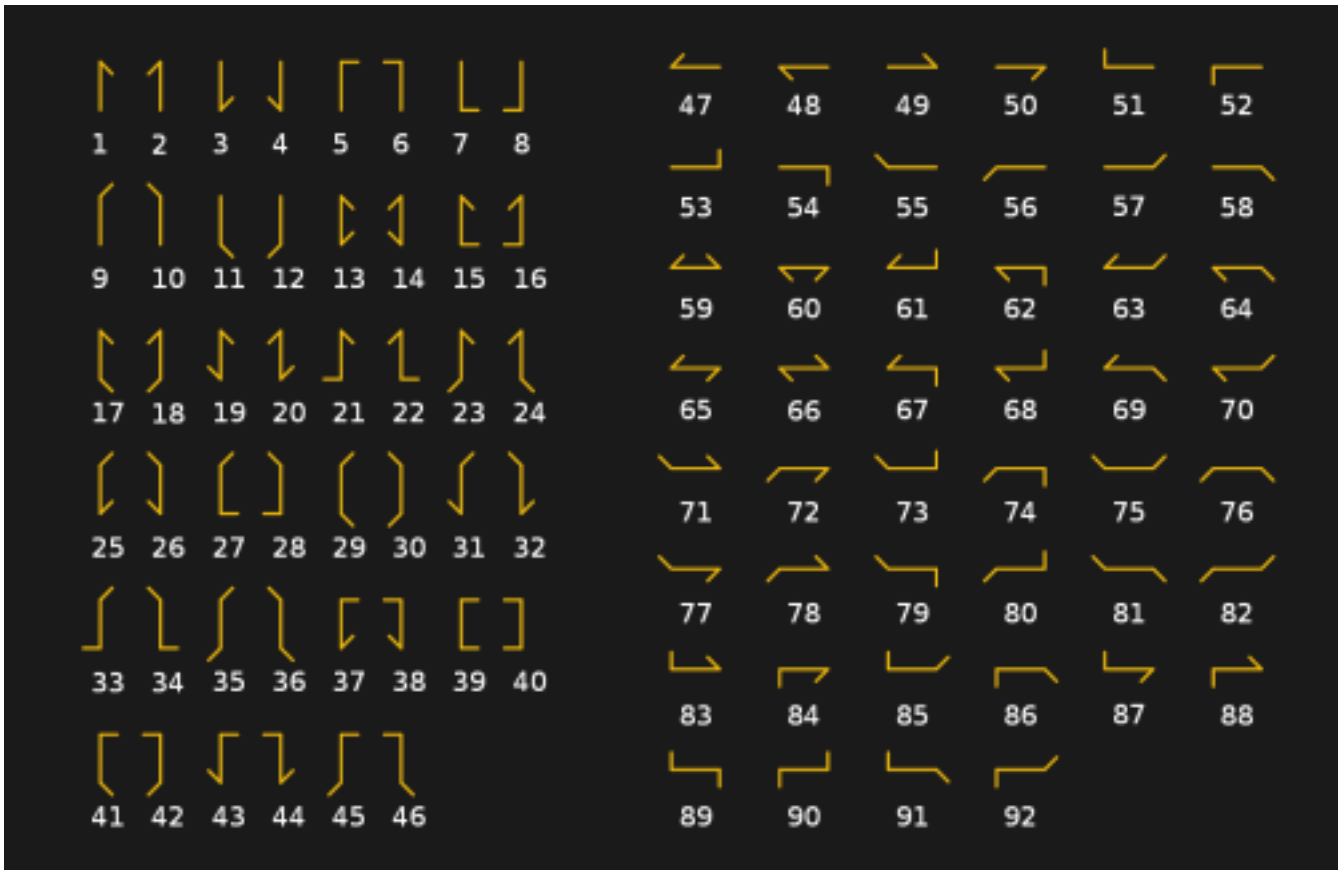
Blocks

A block is a group of characters from a single script, or in some cases, a related group of characters. For example, the Latin block contains all the Latin letters, and the Cyrillic block contains all the Cyrillic letters.



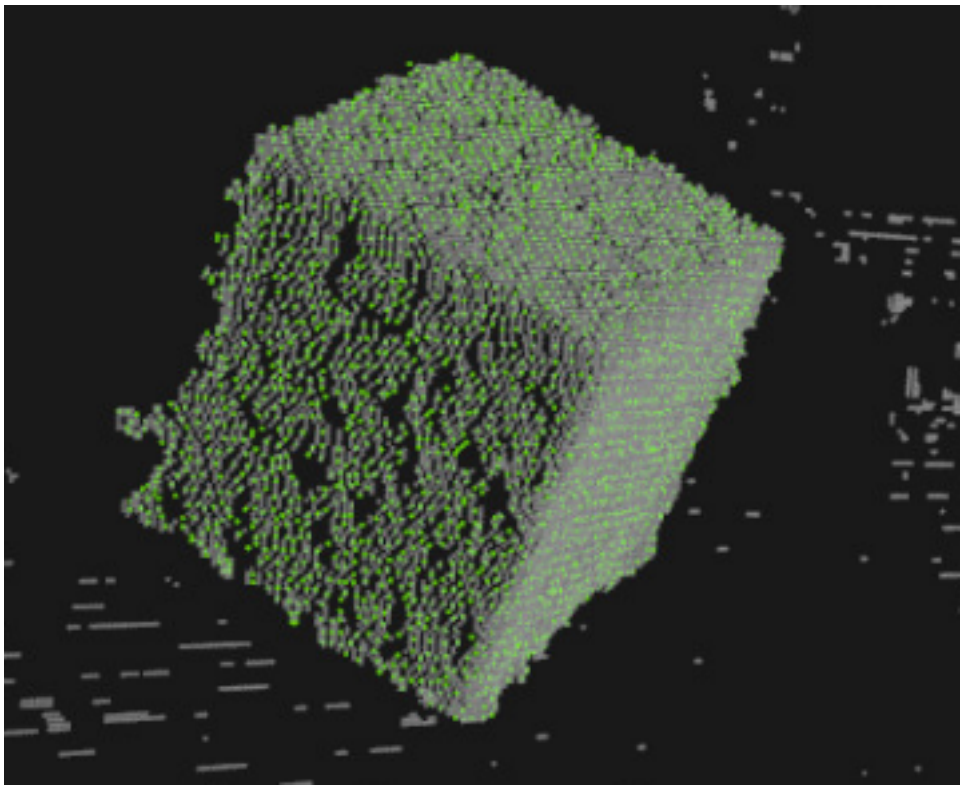
Character position

The position of a character in the Unicode Character Table has no meaning. This means that the character "A" is just as important as the character "α", even though "A" is at a lower code point.



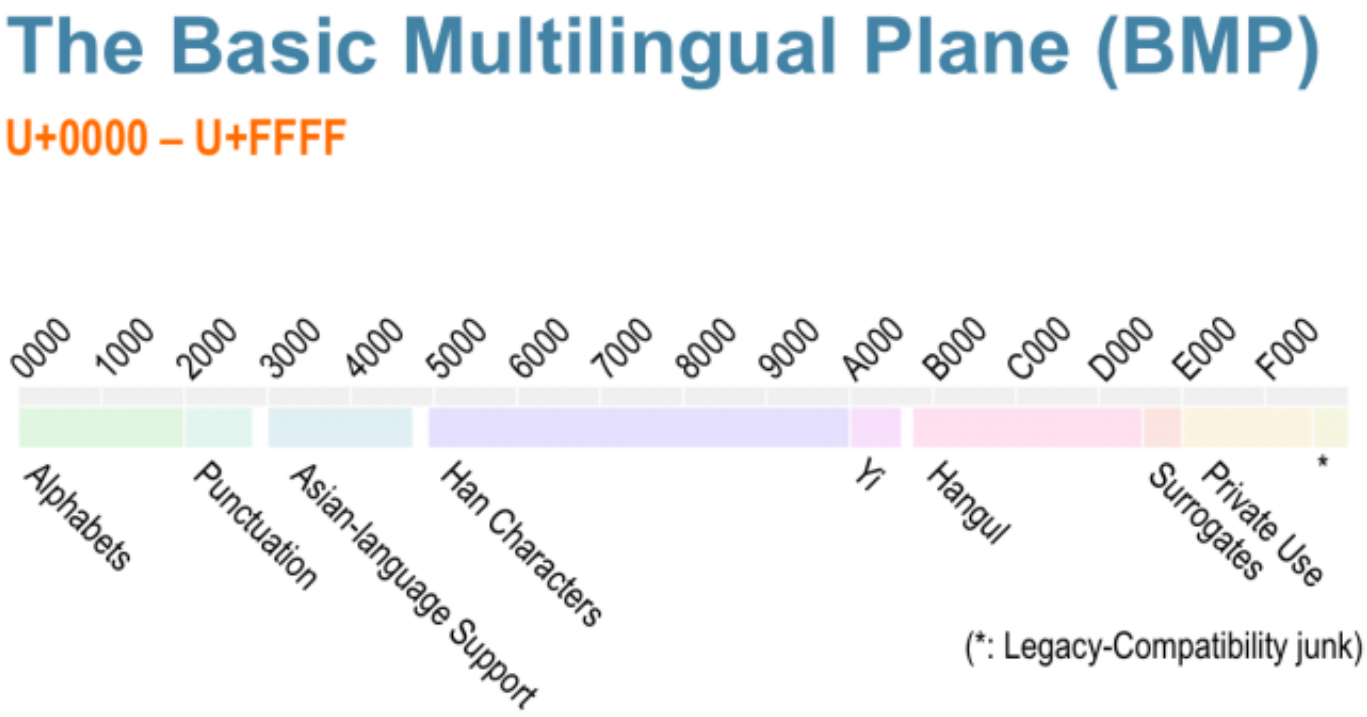
Code point assignment

In theory, the code point of a character only needs to be unique. However, in practice, the code points are assigned according to certain conventions. For example, the most common characters are assigned to the lower code points.



Examples

The Basic Multilingual Plane (BMP) is the most important plane in the Unicode standard. It contains the majority of the most common characters, including the Latin, Cyrillic, Arabic, Chinese, Japanese, and Korean characters.



The Supplementary Multilingual Plane (SMP) contains characters that are less common, such as emojis and rare CJK characters.

The **Supplementary Ideographic Plane (SIP)** contains even rarer CJK characters.

The **Supplementary Special-purpose Plane (SSP)** contains format control characters that do not fit into the BMP.

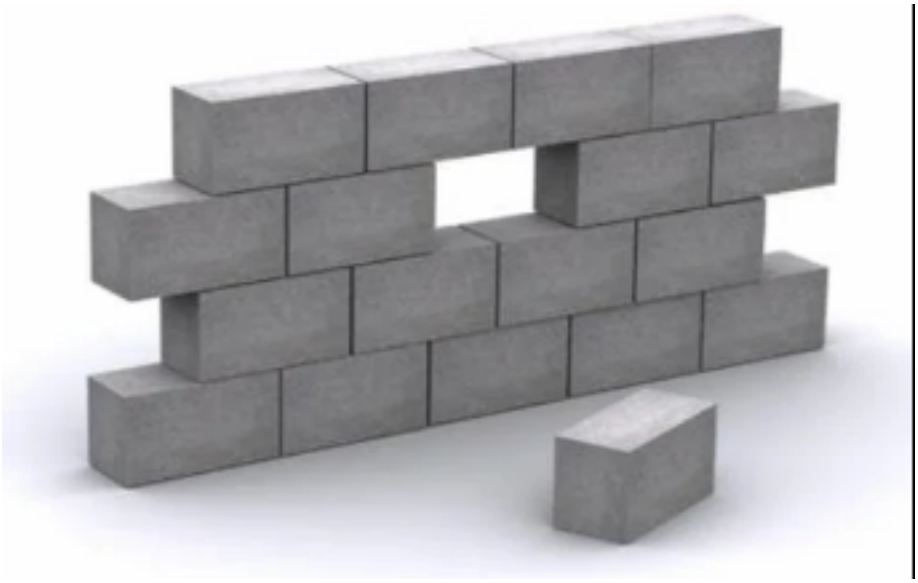
The **Private Use Planes** are reserved for private use. This means that the sender and receiver must agree on how to interpret the characters in these planes.

Why is this important?

Understanding the Unicode Character Table is important for programmers and anyone who works with text. It can help you to troubleshoot problems with text encodings and to ensure that your text is displayed correctly on all devices.

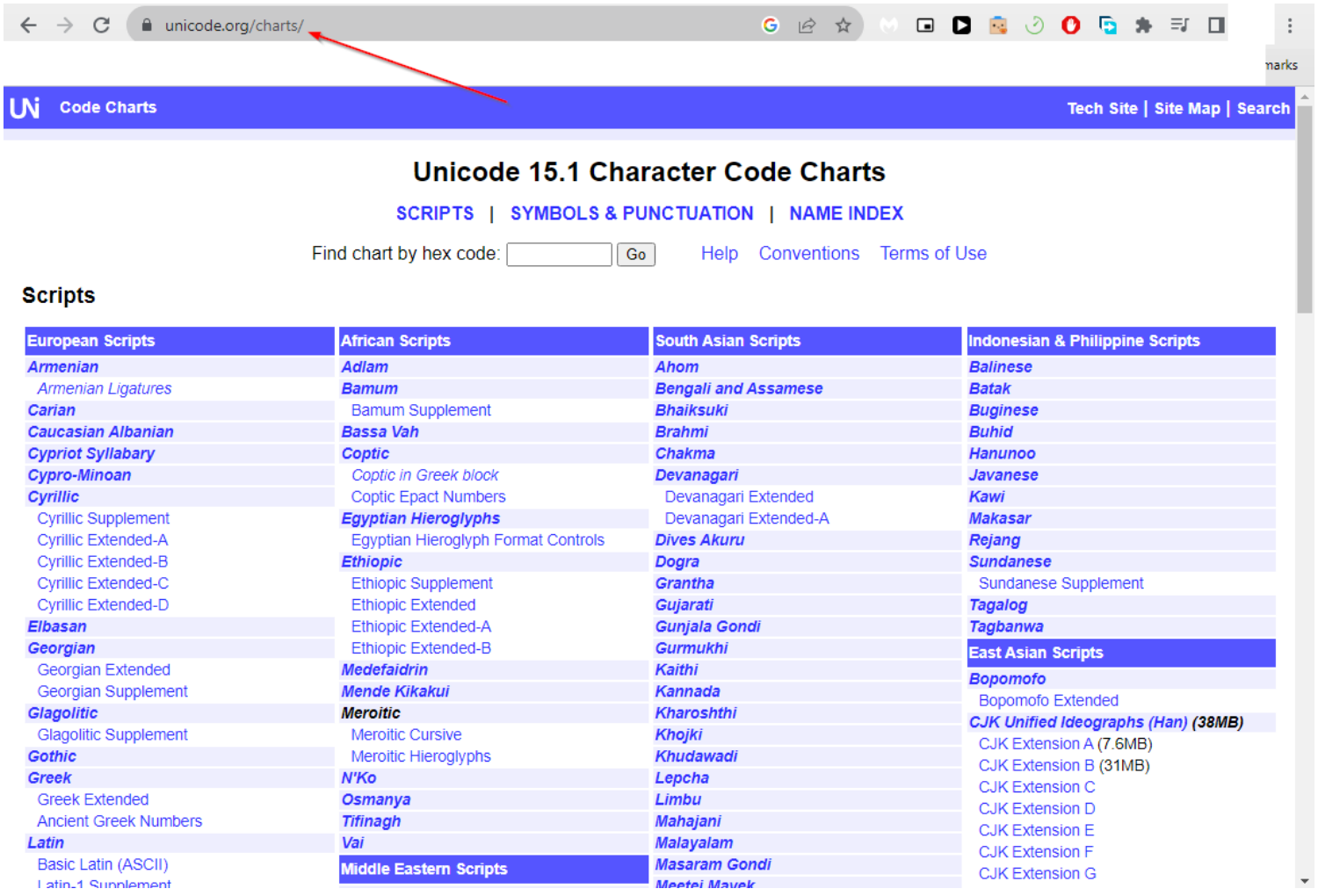
The **Unicode Code Charts** are a set of charts that list all the characters in the Unicode standard. The charts are organized by blocks and scripts.

Blocks are groups of characters from a single script, or in some cases, a related group of characters. For example, the Latin block contains all the Latin letters, and the Cyrillic block contains all the Cyrillic letters.



Scripts are groups of characters that share a common writing system. For example, the Latin script is used to write English, French, Spanish, and many other languages. The Cyrillic script is used to write Russian, Ukrainian, Bulgarian, and many other languages.

The **Unicode Code Charts are organized** in this way so that you can quickly jump to the given script of your language. For example, if you want to find the character for the Greek letter alpha, you can go to the Greek block of the Unicode Code Charts.



The **full Unicode Code Charts** are available in PDF format. However,

the PDF file is 2684 pages long and 110 MB in size. Therefore, it is only recommended to download the full code charts if you need to access them frequently.

Here is an example of how to use the Unicode Code Charts:

- Go to the Unicode Code Charts website.
- Click on the "Blocks" link in the navigation bar.
- Select the block that contains the script of your language.
- Find the character that you are looking for.
- The code point for the character will be listed next to it.
- You can also use the Unicode Code Charts to search for characters by name. To do this, click on the "Search" link in the navigation bar and enter the name of the character that you are looking for.

THE UNICODE CHARACTER DATABASE

The **Unicode Character Database (UCD)** is a collection of documentation and data files that define the character properties for all Unicode characters. The UCD contains over 100 character properties, including:

Name: The name of the character, such as "A" or "alpha".

General category: The primary use of the character, such as letter, number, symbol, or punctuation.

Characteristics: Other general characteristics of the character, such as whitespace, dash, ideographic, alphabetic, or non-character.

Display-related properties: Properties that affect how the character is displayed on screen, such as bidirectional class, shaping, mirroring, and width.

Case: The case of the character, such as upper, lower, title, or folding.

Script and block: The script and block that the character belongs to.

The UCD is a valuable resource for programmers, web developers, and anyone else who works with text. It can be used to determine the properties of a character, to find characters that share certain properties, and to implement functions that work with text.

Here are some examples of how the UCD can be used:

- A programmer can use the UCD to determine if a character is a digit, so that they can implement a function that converts text to numbers.
- A web developer can use the UCD to determine the bidirectional class of a character, so that they can display text correctly in languages that are written from right to left.
- A linguist can use the UCD to find all the characters that belong to a particular script, so that they can study the writing system of that language.

The UCD is a complex resource, but it is an essential tool for anyone who works with text.

Here is an example of how to visualize the character properties from the Unicode Character Database using the website unicode-table.com:

- Go to the unicode-table.com website.
- Enter the name of the character that you want to view.
- Click on the "Search" button.
- The character properties will be displayed in a table.
- For example, if you enter the name "A", the table will show that the character "A" is a letter, that it is uppercase, and that it belongs to the Latin script.

THE UNICODE STANDARD

The Unicode Standard defines three encoding forms for Unicode characters: UTF-8, UTF-16, and UTF-32. These encoding forms use different numbers of bits to represent Unicode characters, as shown in the following table:

Encoding form	Bits per character
UTF-8	8, 16, 24, or 32 bits
UTF-16	16 or 32 bits
UTF-32	32 bits

All three encoding forms can be used to represent the full range of Unicode characters, which includes over 1 million characters. However, they differ in their efficiency and compatibility.

UTF-8 is the most efficient encoding form, and it is also the most widely used encoding form on the internet. UTF-8 is a variable-width encoding, which means that it uses different numbers of bytes to represent different characters. This makes it very efficient for representing the most common characters, such as English letters and numbers.

UTF-16 is a fixed-width encoding form, which means that it uses the same number of bytes to represent all characters. This makes it simpler to implement, but it is less efficient than UTF-8. UTF-16 is commonly used in Windows operating systems and in some programming languages, such as Java.

UTF-32 is also a fixed-width encoding form, and it is the least efficient of the three encoding forms. UTF-32 is rarely used, but it is sometimes used in applications where it is important to be able to represent all Unicode characters without any ambiguity.

The following table shows the advantages and disadvantages of each encoding form:

Encoding form	Advantages	Disadvantages
UTF-8	Most efficient	Not as compatible with older systems
UTF-16	Simple to implement	Not as efficient as UTF-8
UTF-32	Can represent all Unicode characters without ambiguity	Least efficient encoding form

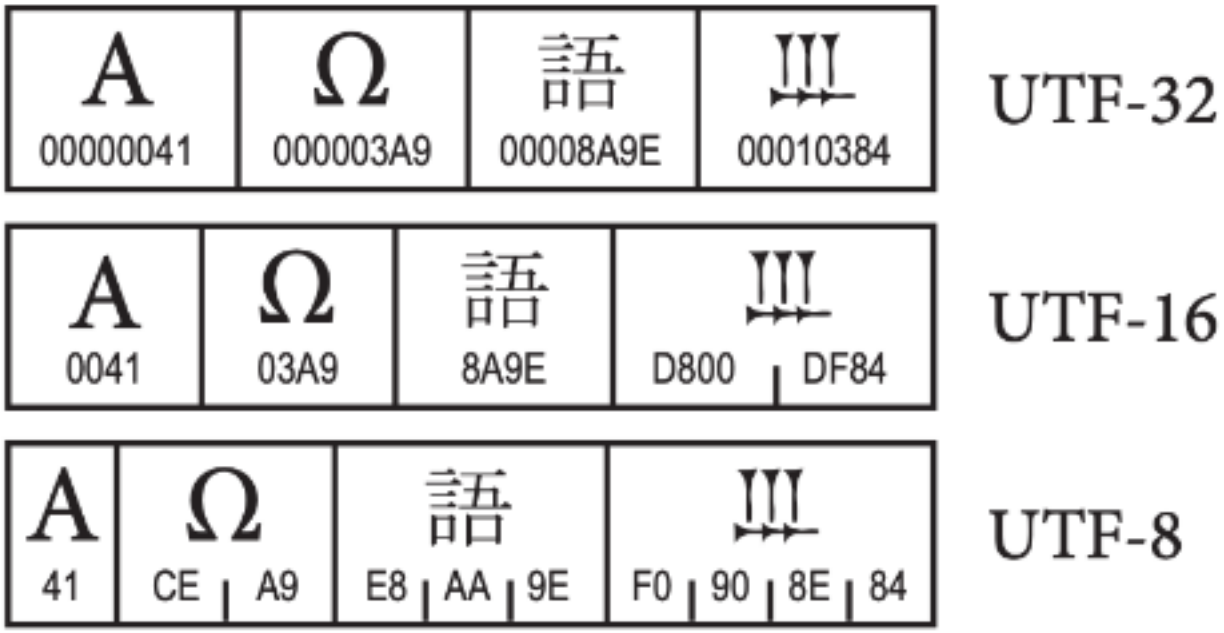
When choosing an encoding form, it is important to consider the following factors:

Efficiency: How important is it to use an efficient encoding form? If efficiency is a top priority, then UTF-8 is the best choice.

Compatibility: How important is it to be compatible with older systems? If compatibility is a top priority, then UTF-16 may be a better choice than UTF-8.

Accuracy: How important is it to be able to represent all Unicode characters without any ambiguity? If accuracy is a top priority, then UTF-32 may be the best choice.

In general, UTF-8 is the best choice for most applications. It is efficient, compatible with most systems, and can represent all Unicode characters.

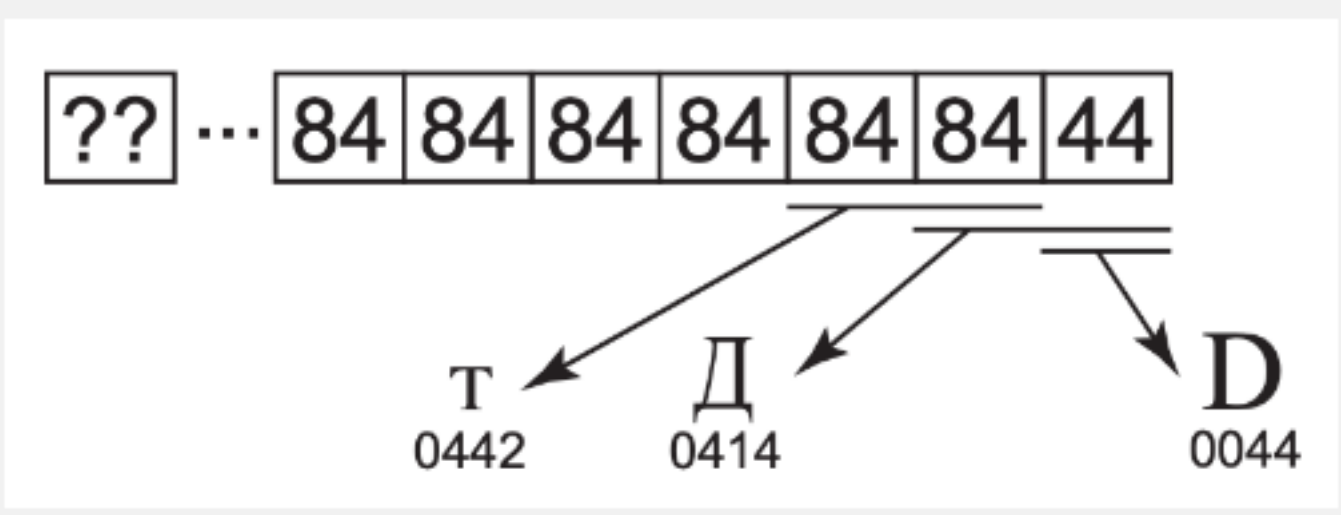


THE PRINCIPLE OF NONOVERLAPPING

The principle of non-overlapping in Unicode encodings states that no byte sequence can be the prefix of another byte sequence. This means that each byte sequence can be uniquely identified by its first byte.

This has several advantages:

- It makes it easier to implement Unicode encoders and decoders.
- It makes it easier to search for and extract characters from text.
- It makes it easier to detect and correct errors in text.



The image above shows an example of how overlapping can lead to ambiguities in legacy encodings.

In the image, the two-byte sequence Д (D) is the trail byte of the three-byte sequence 0044 0414 (D).

If a program is searching for the character "D", it will find the trail byte of the three-byte sequence, but it will not be able to tell if it is the correct match without looking back at the previous two bytes.

This can be difficult and error-prone, especially in large amounts of text.

Unicode encodings avoid this problem by ensuring that no byte sequence is the prefix of another byte sequence.

This means that each byte sequence can be uniquely identified by its first byte. This makes it easy for programs to find, extract, and search for characters in text.

Here is an example of how the principle of non-overlapping is used in UTF-8:

Character	Code point	UTF-8 encoding
A	U+0041	0x41
D	U+0044	0x44
Д	U+0414	0xd0 94

The UTF-8 encoding for the character "Д" is 0xd0 94. This is a two-byte sequence, but neither of the bytes is the prefix of another byte sequence. This means that the program can uniquely identify the character "Д" by its first byte, 0xd0.

The principle of non-overlapping is a key design feature of Unicode encodings. It makes it easier to implement Unicode encoders and decoders, and it makes it easier to search for and extract characters from text.

This table shows the code point and UTF-8 encoding for the characters

"A", "D", and "Д". The code point is the unique identifier for a character in the Unicode standard.

The UTF-8 encoding is a variable-width encoding, which means that it uses different numbers of bytes to represent different characters.

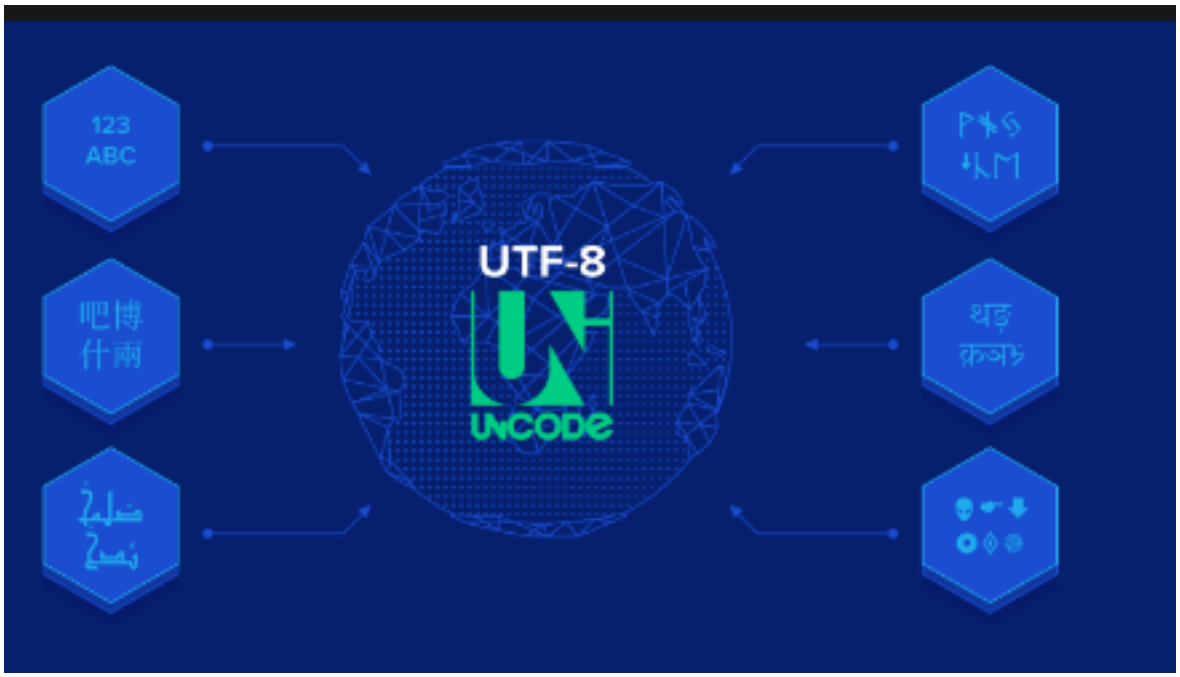
The character "A" is the most common character in the English language.

It is represented by a single byte in UTF-8. The character "D" is also a common character in the English language. It is also represented by a single byte in UTF-8.

The character "Д" is a Cyrillic character. It is represented by two bytes in UTF-8.

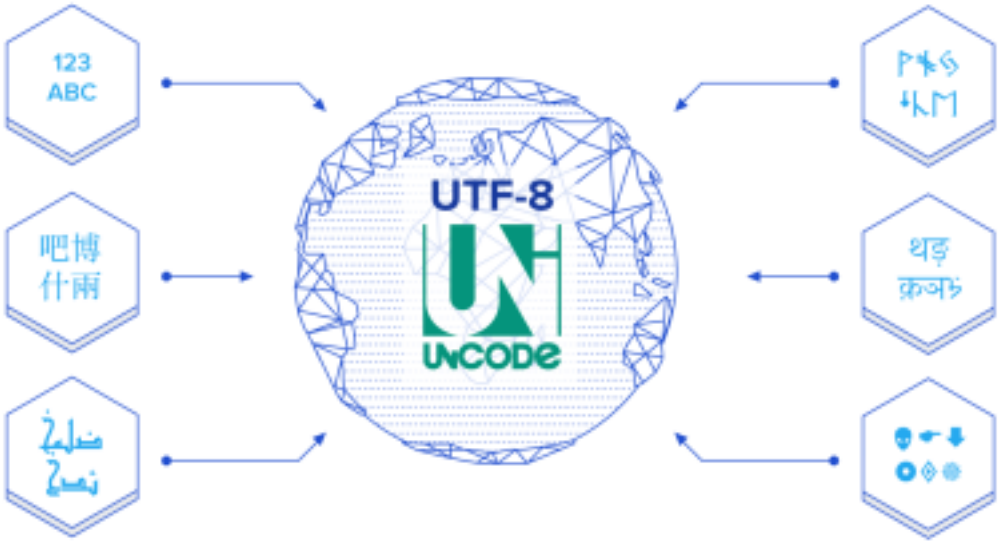
UTF-8

UTF-8 is a variable-width character encoding that is compatible with ASCII. This means that ASCII code points (U+0000..U+007F) are encoded as a single byte in UTF-8, and are therefore indistinguishable from ASCII itself.



UTF-8 is the default Unicode encoding of the web.

UTF-8 is particularly compact when the text contains mainly ASCII characters, but it can be significantly larger than UTF-16 for Asian writing systems, as these characters require three bytes in UTF-8.




```c
#include <stdio.h>
#include <stdlib.h>

unsigned char* encode_utf8(int* codepoints, int count) {
    unsigned char* buf = (unsigned char*)malloc(count * 4); //
Maximum size for UTF-8
    unsigned char* current = buf;

    for (int i = 0; i < count; i++) {
        int codepoint = codepoints[i];

        if (codepoint <= 0x7F) {
            // 1 byte encoding
            *current++ = (unsigned char)codepoint;
        } else if (codepoint <= 0x7FF) {
            // 2 byte encoding
            *current++ = (unsigned char)(0xC0 | (codepoint >> 6));
            *current++ = (unsigned char)(0x80 | (codepoint & 0x3F));
        } else if (codepoint <= 0xFFFF) {
            // 3 byte encoding
            *current++ = (unsigned char)(0xE0 | (codepoint >> 12));
            *current++ = (unsigned char)(0x80 | ((codepoint >> 6) &
0x3F));
            *current++ = (unsigned char)(0x80 | (codepoint & 0x3F));
        } else {
            // 4 byte encoding
            *current++ = (unsigned char)(0xF0 | (codepoint >> 18));
            *current++ = (unsigned char)(0x80 | ((codepoint >> 12) &
0x3F));
            *current++ = (unsigned char)(0x80 | ((codepoint >> 6) &
0x3F));
```

```
            *current++ = (unsigned char)(0x80 | (codepoint & 0x3F));
        }
    }

    return buf;
}

int* decode_utf8(unsigned char* bytes, int length) {
    int* codepoints = (int*)malloc(length); // Maximum size for UTF-8
    int codepointIndex = 0;

    for (int i = 0; i < length;) {
        unsigned char byte = bytes[i];

        if (byte <= 0x7F) {
            // 1 byte encoding
            codepoints[codepointIndex] = byte;
            i++;
        } else if (byte >= 0xC0 && byte <= 0xDF) {
            // 2 byte encoding
            codepoints[codepointIndex] = (byte & 0x1F) << 6;
            i++;
            codepoints[codepointIndex] |= (bytes[i] & 0x3F);
            i++;
        } else if (byte >= 0xE0 && byte <= 0xEF) {
            // 3 byte encoding
            codepoints[codepointIndex] = (byte & 0xF) << 12;
            i++;
            codepoints[codepointIndex] |= (bytes[i] & 0x3F) << 6;
            i++;
            codepoints[codepointIndex] |= (bytes[i] & 0x3F);
            i++;
        } else if (byte >= 0xF0 && byte <= 0xF4) {
            // 4 byte encoding
            codepoints[codepointIndex] = (byte & 0x7) << 18;
            i++;
            codepoints[codepointIndex] |= (bytes[i] & 0x3F) << 12;
            i++;
            codepoints[codepointIndex] |= (bytes[i] & 0x3F) << 6;
            i++;
            codepoints[codepointIndex] |= (bytes[i] & 0x3F);
            i++;
        } else {
            // Invalid UTF-8 encoding
            free(codepoints);
            return NULL;
        }

        codepointIndex++;
    }

    return codepoints;
}
```

```
int main() {
    int codepoints[] = { 'a', 'b', 'c', 0x65E5, 0x672C, 0x8A9E }; //
Example code points
    int count = sizeof(codepoints) / sizeof(codepoints[0]);

    // Encode the code points to UTF-8 bytes.
    unsigned char* utf8_bytes = encode_utf8(codepoints, count);

    // Decode the UTF-8 bytes to code points.
    int* decoded_codepoints = decode_utf8(utf8_bytes, count * 4);

    // Print the original and decoded code points.
    printf("Original Code Points: ");
    for (int i = 0; i < count; i++) {
        printf("U+%04X ", codepoints[i]);
    }
    printf("\n");

    printf("Decoded Code Points: ");
    for (int i = 0; i < count; i++) {
        printf("U+%04X ", decoded_codepoints[i]);
    }
    printf("\n");

    free(utf8_bytes);
    free(decoded_codepoints);

    return 0;
}
```

UTF-8 Encoding and Decoding:

UTF-8 is a variable-length encoding that uses 1 to 4 bytes per character. Encoding in UTF-8 is relatively straightforward:

- Start with a Unicode code point.
- Determine the number of bytes needed based on the code point.
- Write the appropriate header bits (110, 1110, 11110) to indicate the number of bytes.
- Split the code point into 6-bit chunks and distribute them in the remaining bits.
- The character is encoded.

Decoding in UTF-8 is also manageable:

- Read the first byte to determine the number of bytes for the

character.
- Based on the header bits, continue to read the next bytes.
- Reassemble the 6-bit chunks from each byte.
- Combine the chunks to obtain the Unicode code point.
- Repeat for the next character.

UTF-8 is efficient for ASCII characters, as they use only 1 byte. For characters outside the ASCII range, it uses more bytes, with Asian characters typically requiring 3 bytes.

Since we are learning C and WinAPI, that's a C program example for encoding and decoding UTF-8. The main function demonstrates how to use these functions with example code points.

UTF-16

UTF-16 is a variable-width encoding scheme that can represent all of the characters in the Unicode standard. It is the most widely used Unicode encoding on Windows and Android systems.

UTF-16 was designed to be backwards compatible with UCS-2, an earlier encoding scheme that only supported 65,536 characters. For this reason, the first 65,536 characters in Unicode are encoded using the same code points as in UCS-2.

The remaining characters in Unicode are encoded using surrogate pairs. A surrogate pair is a pair of 16-bit code units that represent a single character. The first code unit in the pair is called the high surrogate, and the second code unit is called the low surrogate.

Surrogate pairs are used to encode characters outside of the Basic Multilingual Plane (BMP). The BMP is the first 65,536 characters in Unicode, and it contains the most common characters in most languages.

To encode a character outside of the BMP using UTF-16, the character is first converted to a surrogate pair. The high surrogate is encoded using the first range of unused code points in UTF-16 (0xD800..0xDBFF). The low surrogate is encoded using the second range of unused code points in UTF-16 (0xDC00–0xDFFF).

To decode a surrogate pair, the high and low surrogates are first identified. The high surrogate is then shifted to the left by 10 bits, and the low surrogate is shifted to the right by 10 bits. The

two shifted values are then added together to get the original code point for the character.

UTF-16 is a complex encoding scheme, but it is necessary to support the full range of Unicode characters. It is also widely used on popular operating systems, which makes it a good choice for many applications.

Here is an example of how a surrogate pair is used to encode the Chinese character "中華":

Character	Code point	UTF-16 encoding
中華	U+4E2D	D842 DF63

The character "中華" is outside of the BMP, so it is encoded using a surrogate pair. The high surrogate is D842, and the low surrogate is DF63.

This is a simple table showing the character, code point, and UTF-16 encoding for the Chinese character "中華".

UTF-16 is a variable-width encoding, which means that the number of bytes used to encode a character can vary depending on the character. The character "中華" is encoded using a surrogate pair, which is two 16-bit code units. The high surrogate is D842, and the low surrogate is DF63.

Here is a table showing the binary representation of the UTF-16 encoding for the character "中華":

Byte	Binary representation
D8	11011000
42	01000010
DF	11011111
63	01100011

To encode a character in UTF-16, we first need to determine if the character is in the Basic Multilingual Plane (BMP). The BMP is the first 65,536 characters in Unicode, and it contains the most common characters in most languages.

If the character is in the BMP, then we can simply encode it as a single 16-bit code unit. To do this, we simply convert the character's code point to a 16-bit integer.

If the character is not in the BMP, then we need to encode it as a surrogate pair. A surrogate pair is two 16-bit code units that represent a single character.

To encode a surrogate pair, we first subtract 0x10000 from the character's code point. This gives us a 20-bit number.

We then split the 20-bit number into two 10-bit numbers. The first 10-bit number is the high surrogate, and the second 10-bit number is the low surrogate.

We then add 0xD800 to the high surrogate and 0xDC00 to the low surrogate. This gives us the two 16-bit code units that represent the surrogate pair.

Here is a table showing how to encode different types of characters in UTF-16:

Character type	Encoding
BMP character	Single 16-bit code unit
Non-BMP character	Surrogate pair

Preferred Usage

UTF-16 is a balanced representation that is reasonably compact as all the common, heavily used characters fit into a single 16-bit code unit. This encoding is often used by programming languages as their internal representation of strings for that reason.

However, for file encoding, UTF-8 is by far the most privileged encoding. UTF-8 is a variable-width encoding, which means that it uses different numbers of bytes to represent different characters. This makes it more efficient for storing text that contains a mix of languages.

To code in C for UTF-16, you can use the following steps:

Include the uchar.h header file. This header file contains the definition of the char16_t data type, which is used to represent UTF-16 code units.

Declare your variables and strings using the char16_t data type.

Use the u prefix to indicate that a string literal is in UTF-16 encoding. For example, the following string literal is in UTF-16 encoding:

```
char16_t my_string[] = u"This is a UTF-16 string.";
```

Use the standard C library functions to manipulate UTF-16 strings. For example, the following code uses the strlen() function to get the length of a UTF-16 string:

```
size_t string_length = strlen(my_string);
```

Use the printf() function to print UTF-16 strings to the console. For example, the following code prints the my_string variable to the console:

```
printf("%s\n", my_string);
```

Here is an example of a complete C program that codes for UTF-16:

```
#include <uchar.h>

int main() {
  char16_t my_string[] = u"This is a UTF-16 string.";

  ;Get the length of the string.
  size_t string_length = strlen(my_string);

  ;Print the string to the console.
  printf("%s\n", my_string);

  return 0;
}
```

This program will print the following output to the console:

```
This is a UTF-16 string.
```

You can also use C libraries to perform more complex operations on UTF-16 strings, such as encoding and decoding UTF-16 strings, and converting UTF-16 strings to other encodings.

Here is the C program for UTF-16 encoding and decoding in C:

```c
#include <stdio.h>
#include <stdlib.h>

unsigned short* encode_utf16(int* codepoints, int count) {
    unsigned short* buf = (unsigned short*)malloc(count * 2); // 
Maximum size for UTF-16
    unsigned short* current = buf;

    for (int i = 0; i < count; i++) {
        int codepoint = codepoints[i];

        if (codepoint <= 0xFFFF) {
            // 2-byte encoding (BMP characters)
            *current++ = (unsigned short)codepoint;
        } else if (codepoint <= 0x10FFFF) {
            // 4-byte encoding (Supplementary Planes)
            codepoint -= 0x10000;
            *current++ = (unsigned short)(((codepoint >> 10) & 0x3FF) 
+ 0xD800);
            *current++ = (unsigned short)((codepoint & 0x3FF) + 
0xDC00);
        } else {
            // Invalid code point
            free(buf);
            return NULL;
        }
    }

    return buf;
}

int* decode_utf16(unsigned short* utf16, int length) {
    int* codepoints = (int*)malloc(length); // Maximum size for 
UTF-16
    int codepointIndex = 0;
    int i = 0;

    while (i < length) {
        unsigned short utf16_char = utf16[i];

        if (utf16_char >= 0xD800 && utf16_char <= 0xDBFF) {
            // 4-byte encoding (Surrogate pair)
            if (i + 1 < length) {
                unsigned short low_surrogate = utf16[i + 1];

                if (low_surrogate >= 0xDC00 && low_surrogate <= 
0xDFFF) {
                    int codepoint = ((utf16_char - 0xD800) << 10) + 
(low_surrogate - 0xDC00) + 0x10000;
                    codepoints[codepointIndex] = codepoint;
```

```
                    codepointIndex++;
                    i += 2;
                } else {
                    // Invalid low surrogate
                    free(codepoints);
                    return NULL;
                }
            } else {
                // Incomplete surrogate pair
                free(codepoints);
                return NULL;
            }
        } else {
            // 2-byte encoding (BMP character)
            codepoints[codepointIndex] = utf16_char;
            codepointIndex++;
            i++;
        }
    }

    return codepoints;
}

int main() {
    int codepoints[] = { 'a', 'b', 'c', 0x65E5, 0x672C, 0x8A9E }; //
Example code points
    int count = sizeof(codepoints) / sizeof(codepoints[0]);

    // Encode the code points to UTF-16.
    unsigned short* utf16 = encode_utf16(codepoints, count);

    // Decode the UTF-16 to code points.
    int* decoded_codepoints = decode_utf16(utf16, count * 2);

    // Print the original and decoded code points.
    printf("Original Code Points: ");
    for (int i = 0; i < count; i++) {
        printf("U+%04X ", codepoints[i]);
    }
    printf("\n");

    printf("Decoded Code Points: ");
    for (int i = 0; i < count; i++) {
        printf("U+%04X ", decoded_codepoints[i]);
    }
    printf("\n");

    free(utf16);
    free(decoded_codepoints);

    return 0;
}
```

UTF-16 Encoding and Decoding:

UTF-16 uses a fixed 2-byte or 4-byte encoding. Encoding UTF-16 is relatively simple:

- Start with a Unicode code point.
- Check if the code point fits in a 2-byte unit (Basic Multilingual Plane, BMP) or requires a surrogate pair (Supplementary Planes).
- For BMP characters, encode them as 2 bytes. For Supplementary Plane characters, split into high and low surrogates and encode each.

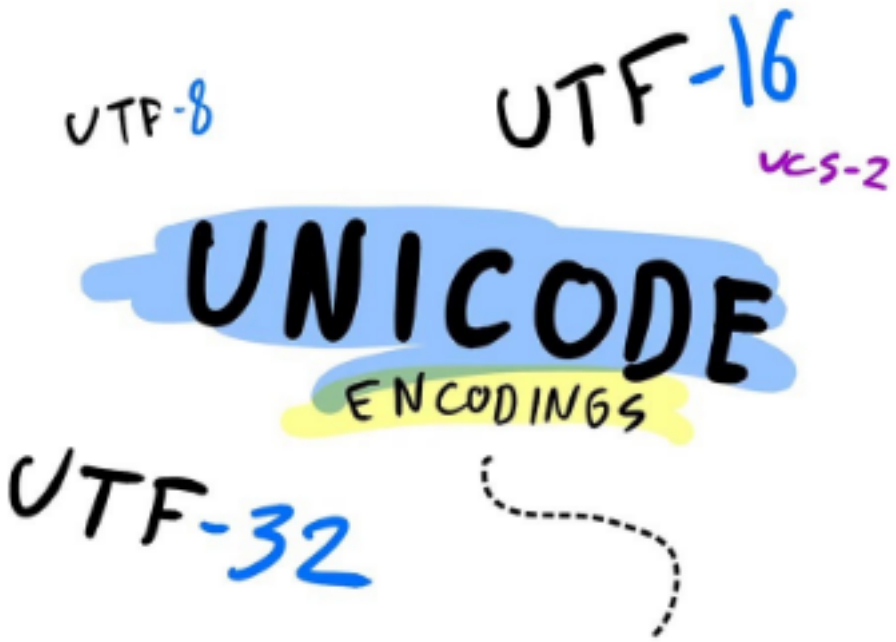
Decoding UTF-16 is also straightforward:

- Read 2 bytes (a unit) or 4 bytes (surrogate pair).
- Combine the bytes to obtain the code point.
- Repeat for the next unit or pair.

UTF-16 is efficient for BMP characters but less so for Supplementary Plane characters due to the surrogate pair requirement.

UTF-32

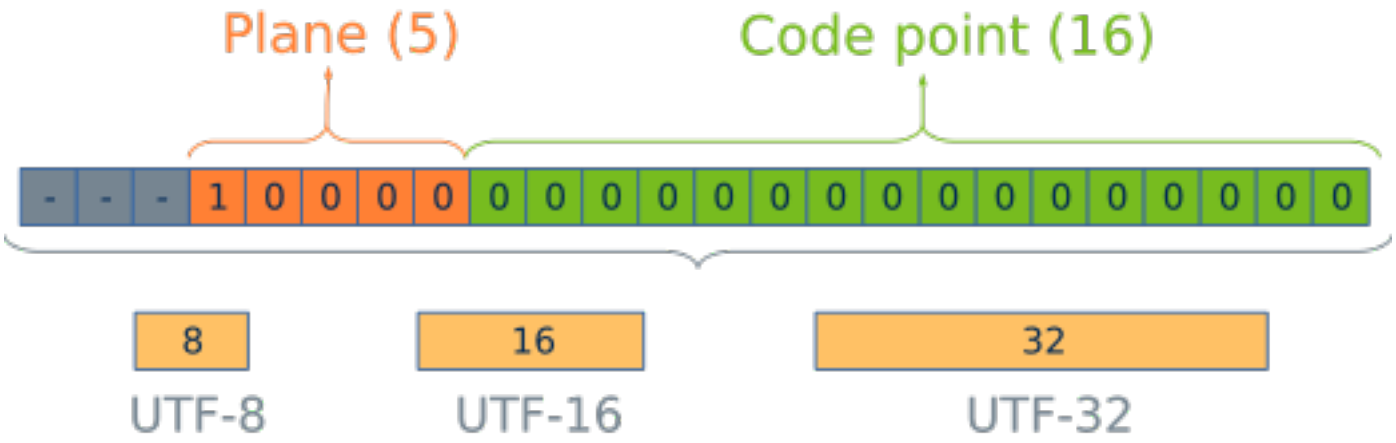
UTF-32 is a fixed-width character encoding form, which means that each Unicode code point is represented by a single 32-bit code unit. This makes UTF-32 the simplest Unicode encoding form to implement, but it also makes it the most inefficient.



UTF-32 is rarely used in practice because it is so large. However, it

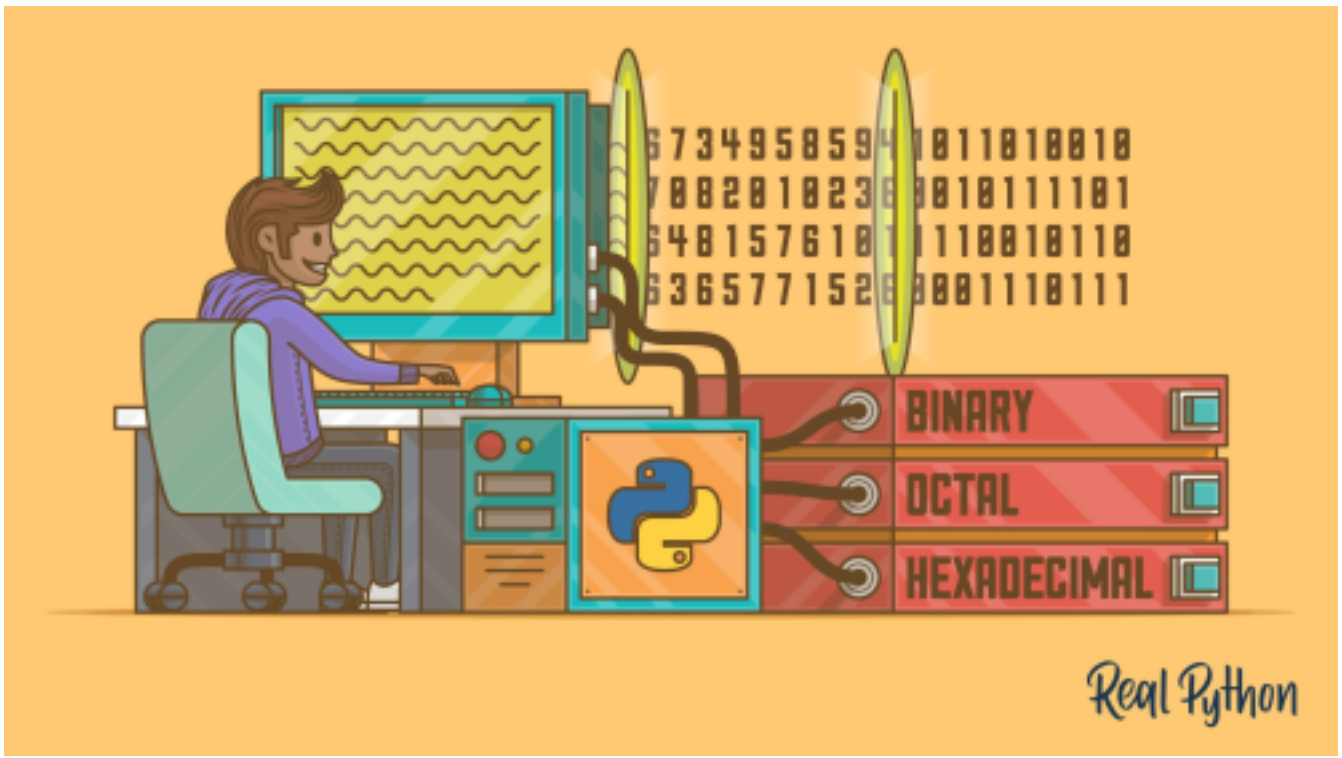
is sometimes used in applications where it is important to be able to represent all Unicode characters without any ambiguity, such as in text processing libraries and operating systems.

Here is a rewritten version of your explanation:



UTF-32 is the simplest way to encode Unicode characters, but it is also the least efficient. This is because each character is represented by a single 32-bit code unit, which is more space than is needed for most characters.

UTF-32 is rarely used in practice, but it is sometimes used in applications where it is important to be able to represent all Unicode characters without any ambiguity. For example, the first version of Python 3 represented strings as sequences of Unicode code points, but Python 3.3 changed the implementation to optimize the memory usage.



Another example is the Unicode Consortium's own documentation, which uses UTF-32 to represent all characters. This makes it possible to search and navigate the documentation for any character, regardless of its script or language.

Overall, UTF-32 is a simple but inefficient way to encode Unicode characters. It is rarely used in practice, but it is sometimes used in applications where it is important to be able to represent all Unicode characters without any ambiguity.

```
#include <stdio.h>
#include <stdlib.h>

// Function to encode Unicode code points to UTF-32
unsigned int* encode_utf32(int* codepoints, int count) {
    unsigned int* buf = (unsigned int*)malloc(count * 4); // Maximum
size for UTF-32
    unsigned int* current = buf;

    for (int i = 0; i < count; i++) {
        int codepoint = codepoints[i];
        if (codepoint <= 0x10FFFF) {
            *current++ = (unsigned int)codepoint;
        } else {
            // Invalid code point
            free(buf);
            return NULL;
        }
    }

    return buf;
}

// Function to decode UTF-32 to Unicode code points
int* decode_utf32(unsigned int* utf32, int length) {
    int* codepoints = (int*)malloc(length); // Maximum size for
UTF-32
    int codepointIndex = 0;

    for (int i = 0; i < length; i++) {
        int utf32_char = utf32[i];
        if (utf32_char <= 0x10FFFF) {
            codepoints[codepointIndex] = utf32_char;
            codepointIndex++;
        } else {
            // Invalid code point
            free(codepoints);
            return NULL;
        }
```

```
    }

    return codepoints;
}

int main() {
    int codepoints[] = { 'a', 'b', 'c', 0x65E5, 0x672C, 0x8A9E }; //
Example code points
    int count = sizeof(codepoints) / sizeof(codepoints[0]);

    // Encode the code points to UTF-32.
    unsigned int* utf32 = encode_utf32(codepoints, count);

    // Decode the UTF-32 to code points.
    int* decoded_codepoints = decode_utf32(utf32, count);

    // Print the original and decoded code points.
    printf("Original Code Points: ");
    for (int i = 0; i < count; i++) {
        printf("U+%04X ", codepoints[i]);
    }
    printf("\n");

    printf("Decoded Code Points: ");
    for (int i = 0; i < count; i++) {
        printf("U+%04X ", decoded_codepoints[i]);
    }
    printf("\n");

    free(utf32);
    free(decoded_codepoints);

    return 0;
}
```

UTF-32 Encoding and Decoding:

UTF-32 uses a fixed 4-byte encoding. Encoding UTF-32 is the simplest:

- Start with a Unicode code point.
- Encode it as 4 bytes directly.

Decoding UTF-32 is as simple as encoding:

- Read 4 bytes to obtain the code point.
- Repeat for the next code point.

UTF-32 is the most straightforward encoding but is less space-efficient, as it uses 4 bytes for every character.

In practice:

UTF-8 has become popular on the web because it's compact for ASCII characters and handles various scripts efficiently.

UTF-16 is used in some programming environments and is more efficient for BMP characters.

UTF-32 is less common due to its space inefficiency but simplifies encoding and decoding. The choice depends on the specific requirements of the application.

THE IMPLEMENTATION

To support Unicode, a program must be able to:

- Read and write Unicode data. This means understanding how Unicode characters are encoded and decoded in different formats, such as UTF-8, UTF-16, and UTF-32.
- Store Unicode data internally. This means using data structures that can represent the full range of Unicode characters.
- Display Unicode data to the user. This means using fonts and rendering systems that can support the full range of Unicode characters.

For an operating system, supporting Unicode means providing the following functionality:

- A Unicode character encoding. The operating system must have a default Unicode character encoding that is used for all text processing.
- Unicode fonts. The operating system must provide a set of Unicode fonts that can be used to display text to the user.
- Unicode input methods. The operating system must provide a way for users to input text in different languages and writing systems.

For a programming language, supporting Unicode means providing the following functionality:

• Unicode string types. The programming language must have string types that can represent the full range of Unicode characters.
• Unicode functions. The programming language must provide functions for encoding and decoding Unicode data, and for converting between different Unicode character encodings.

For a desktop application, supporting Unicode means:

• Using the Unicode functionality provided by the operating system. This includes using the Unicode character encoding, Unicode fonts, and Unicode input methods.

For a web application, supporting Unicode:

• Means using the Unicode functionality provided by the web browser. This includes using the Unicode character encoding and Unicode fonts.

Here is a simplified explanation of the role of Unicode in each step of a program:

• **Input:** When a user inputs text, the program must convert it to a Unicode representation.
• **Processing:** When the program processes text, it must use a Unicode representation of the text.
• **Output:** When the program outputs text, it must convert it from a Unicode representation to the appropriate encoding for the output device.
• **Unicode** is essential for supporting a wide range of languages and writing systems in modern software. By implementing Unicode support, programs can be used by people all over the world.

****

I'LL STOP THERE, VISIT THAT PDF ON
UNICODE IMPLEMENTATION FOR ADVANCED PROGRAMMERS.

****