# Información Digital
## Representación y Codificación

El texto que sigue ha sido seleccionado por el profesorado como una lectura recomendada para el MOOC “Información Digital”, organizado por la Universidad de Granada.

***https://betterexplained.com/articles/unicode/***

# *Unicode and You*

I'm a Unicode newbie. But like many newbies, I had an urge to learn once my interest was piqued by an introduction to Unicode.

Unicode isn't hard to understand, but it does cover some low-level CS concepts, like byte order. Reading about Unicode is a nice lesson in design tradeoffs and backwards compatibility.

My thoughts are below. Read them alone, or as a follow-up to Joel's unicode article above. If you're like me, you'll get an itch to read about the details in the Unicode specs or in Wikipedia. Really, it can be cool, I swear.

## *Key concepts*

Let's level set on some ideas:

**Ideas and data are different.** The concept of "A" is something different than marks on paper, the sound "aaay" or the number 65 stored inside a computer.

**One idea has many possible encodings.** An encoding is just a method to transform an idea (like the letter "A") into raw data (bits and bytes). The idea of "A" can be encoded many different ways. Encodings differ in efficiency and compatibility.

**Know thy encoding.** When reading data, you must know the encoding used in order to interpret it properly. This is a simple but important concept. If you see the number 65 in binary, what does it really mean? "A" in ASCII? Your age? Your IQ? Unless there is some context, you'd never know. Imagine if someone came up to you and said "65". You'd have no idea what they were talking about. Now imagine they came up and said "The following number is an ASCII character: 65". Weird, yes, but see how much clearer it is?

Embrace the philosophy that a concept and the data that stores it are different. Let it rustle around in your mind…

Got it? Let's dive in.

## *Back to ASCII and Code Pages*

You've probably heard of the ASCII/ANSI characters sets. They map the numeric values 0-127 to various Western characters and control codes (newline, tab, etc.). Note that values 0-127 fit in the lower 7 bits in an 8-bit byte. ASCII does not explicitly define what values 128-255 map to.

Now, ASCII encoding works great for English text (using Western characters), but the world is a big place. What about Arabic, Chinese and Hebrew?

To solve this, computer makers defined "code pages" that used the undefined space from 128-255 in ASCII, mapping it to various characters they needed. Unfortunately, 128 additional characters aren't enough for the entire world: code pages varied by country (Russian code page, Hebrew code page, etc.).

If people with the same code page exchanged data, all was good. Character #200 on my machine was the same as Character #200 on yours. But if codepages mixed (Russian sender, Hebrew receiver), things got strange.

The character mapped to #200 was different in Russian and Hebrew, and you can imagine the confusion that caused for things like email and birthday invitations. It's a big IF whether or not someone will read your message using the same codepage you authored your text. If you visit an international website, for example, your browser could try to *guess* the codepage if it was not specified ("Hrm… this text has a lot of character #213 and #218… probably Hebrew"). But clearly this method was error-prone: codepages needed to be rescued.

## *Unicode to the Rescue*

The world had a conundrum: they couldn't agree on what numbers mapped to what letters in ASCII. The Unicode group went back to the basics: Letters are abstract concepts. Unicode labeled each abstract character with a "code point". For example, "A" mapped to code point U+0041 (this code point is in hex; code point 65 in decimal).

The Unicode group did the hard work of mapping each character in every language to some code point (not without fierce debate, I am sure). When all was done, the Unicode standard left room for over 1 million code points, enough for all known languages with room to spare for undiscovered civilizations. For fun, you can browse the codepoints with the charmap utility (Start Menu > Run > Charmap) or online at Unicode.org.

This brings us to our first design decision: **compatibility**.

For compatibility with ASCII, code points U+0000 to U+007F (0-127) were the same as ASCII. Purists probably didn't like this, because the full Latin character sets were defined elsewhere, and now one letter had 2 codepoints. Also, this put Western characters "first", whereas Chinese, Arabic and the "nonstandard" languages were stuck in the non-sleek codepoints that require 2 bytes to store.

However, this design was necessary – ASCII was a standard, and if Unicode was to be adopted by the Western world it needed to be compatible, without question. Now, the majority of common languages fit into the first 65535 codepoints, which can be stored as 2 bytes.

Phew. The world was a better place, and everyone agreed on what codepoint mapped to what character.

But the question remained: How do we store a codepoint as data?

### *Encoding to the Rescue*

From above, encoding turns an idea into raw data. In this case, the idea is a codepoint.

For example, let's look at the ASCII "encoding" scheme to store Unicode codepoints. The rules are pretty simple:

- Code points from U+0000 to U+007F are stored in a single byte
- Code points above U+0080 are dropped on the floor, never to be seen again

Simple, right?

As you can see, ASCII isn't great for storing Unicode – in fact, it ignores most Unicode codepoints altogether. If you have a Unicode document and save it as ASCII -wham- all your special characters are gone. You'll often see this as a warning in some text editors when you save Unicode data in a file original saved as ASCII.

But the example has a purpose. An encoding is a system to convert an idea into data. In this case, the conversion can be politely called "lossy".

I did Unicode experiments with Notepad (can read/write Unicode) and Programmer's Notepad, a hex editor. I wanted to see the raw bytes that notepad was saving. To the examples for yourself:

- Open notepad and type "Hello"
- Save file separately as ANSI, Unicode, Unicode Big Endian, UTF-8

- Open file with Programmer's Notepad and do View > View Hex

### *All about ASCII*

Let's write "Hello" in notepad, save as ANSI (ASCII) and open it in a hex editor. It looks like this:

```
Byte:     48 65 6C 6C 6F
Letter:   H  e  l  l  o
```

ASCII is important because many tools and communication protocols only accept ASCII characters. It's a generally accepted minimum bar for text. Because of its universal acceptance, some Unicode encodings will transform codepoints into series of ASCII characters so they can be transmitted without issue.

Now, in the example above, we know the data is text because we authored it. If we randomly found the file, we could *assume* it was ASCII text given its contents, but it might be an account number or other data for all we know, that happens to look like "Hello" in ASCII.

Usually, we can make a good guess about what data is supposed to be, based on certain headers or "Magic Numbers" (special character sequences) that appear in certain places. But you can never be sure, and sometimes you can guess wrong.

Don't believe me? Ok, do the following

- Open notepad
- Write "this program can break"
- Save the file as "blah.txt" (or anything else
- Open the file in notepad

Wow… whoa… what happened? I'll leave this as an exercise for the reader.

### *UCS-2 / UTF-16*

This is the encoding I first thought of when I heard "Unicode" – store every character as 2 bytes (what a waste!). At a base level, this can handle codepoints 0x0000 to 0xFFFF, or 0-65535 for you humans out there. And 65,535 should be enough characters for anybody (there are ways to store codepoints above 65535, but read the spec for more details).

Storing data in multiple bytes leads to my favorite conundrum: byte order! Some computers store the little byte first, others the big byte.

To resolve the problem, we can do the following:

- Option 1: **Choose a convention** that says all text data must be big or little-endian. This won't happen – computers on the wrong side of the decision would suffer inefficiency every time they opened a file, since they cannot convert it to the other byte order.
- Option 2: **Everyone agrees to a byte order mark (BOM)**, a header at the top of each file. If you open a file and the BOM is backwards, it means it was encoded in a different byte order and needs to be converted.

The solution was the BOM header: UCS-2 encodings could write codepoint U+FEFF as a file header. If you open a UCS-2 string and see FEFF, the data is in the right byte order and can be used directly. If you see FFFE, the data came from another type of machine, and needs to be converted to your architecture. This involves swapping every byte in the file.

But unfortunately, things are not that simple. The BOM is actually a valid Unicode character – what if someone sent a file without a header, and that character was actually part of the file?

This is an open issue in Unicode. The suggestion is to avoid U+FEFF except for headers, and use alternative characters instead (there are equivalents).

This opens up design observation #2: **Multi-byte data will have byte order issues!**

ASCII never had to worry about byte order – each character was a single byte, and could not be misinterpreted. But realistically, if you see bytes 0xFEFF or 0xFFEE at the start of a file, it's a good chance it's a BOM in a Unicode text file. It's probably an indication of byte order. Probably.

(Aside: UCS-2 stores data in a flat 16-bit chunk. UTF-16 allows up to 20 bits split between 2 16-bit characters, known as a surrogate pair. Each character in the surrogate pair is an invalid unicode character by itself, but together a valid one can be extracted.)

### *UCS-2 Example*

Type "Hello" in notepad and save it as Unicode (little-endian UCS-2 is the native format on Windows):

Hello-little-endian:

```
FF FE  4800 6500 6C00 6c00 6F00
header H    e    l    l    o
```

Save it again as Unicode Big Endian, and you get:

Hello-big-endian:

```
FE FF  0048 0065 006C 006C 006F
header H    e    l    l    o
```

Observations

- The header BOM (U+FEFF) shows up as expected: FF FE for little-endian, FEFF for big
- Letters use 2 bytes no matter what: “H” is 0x48 in ASCII, and 0x0048 in UCS-2
- Encoding is simple. Take the codepoint in hex and write it out in 2 bytes. No extra processing is required.
- The encoding is too simple. It wastes space for plain ASCII text that does not use the high-order byte. And ASCII text is very common.
- The encoding inserts null bytes (0x00) which can be a problem. Old-school ASCII programs may think the Unicode string has ended when it gets to the null byte. On a little-endian machine, reading one byte at a time, you’d get to H (H = 0x4800) and then hit the null and stop. On a big endian machine, you’d hit the null first (H is 0x0048) and not even see the H in ASCII. Not good.

Design observation #3 * Consider backwards compatibility. How will an old program read new data? Ignoring new data is good. Breaking on new data is bad.

### *UTF-8*

UCS-2 / UTF-16 is nice and simple, but boy it does waste some bits. Not only does it double ASCII, but the converted ASCII might not even be readable due to the null characters.

Enter UTF-8. Its goal is to encode Unicode characters in single byte where possible (ASCII), and not break ASCII applications by having null characters. It is the default encoding for XML.

Read the UTF-8 specs for more detail, but at a high level:

- Code points 0 – 007F are stored as regular, single-byte ASCII.
- Code points 0080 and above are converted to binary and stored (encoded) in a series of bytes.
- The first “count” byte indicates the number of bytes for the codepoint, including the count byte. These bytes start with 11..0:

  110xxxxx (The leading “11” is indicates 2 bytes in sequence, including the “count” byte)

1110xxxx (1110 -> 3 bytes in sequence)

11110xxx (11110 -> 4 bytes in sequence)

- Bytes starting with 10… are “data” bytes and contain information for the codepoint. A 2-byte example looks like this

  110xxxxx 10xxxxxx

This means there are 2 bytes in the sequence. The X’s represent the binary value of the codepoint, which needs to squeeze in the remaining bits.

Observations about UTF-8

- No null bytes. All ASCII characters (0-127) are the same. Non-ASCII characters all start with “1” as the highest bit.
- ASCII text is stored identically and efficiently.
- Unicode characters start with “1” as the high bit, and can be ignored by ASCII-only programs (however, they may be discarded in some cases! See UTF-7 for more details).
- There is a time-space tradeoff. There is processing to be done on every Unicode character, but this is a reasonable tradeoff.

Design principle #4

- UTF-8 addresses the 80% case well (ASCII), while making the other cases possible (Unicode). UCS-2 addresses all cases equally, but is inefficient in the 80% case for solve for the 99% case. But UCS-2 is less processing-intensive than UTF-8, which requires bit manipulation on all Unicode characters.
- Why does XML store data in UTF-8 instead of UCS-2? Is space or processing power more important when reading XML documents?
- Why does Windows XP store strings as UCS-2 natively? Is space or processing power more important for the OS internals?

In any case, UTF-8 still needs a header to indicate how the text was encoded. Otherwise, it could be interpreted as straight ASCII with some codepage to handle values above 127. It still uses the U+FEFF codepoint as a BOM, but the BOM itself is encoded in UTF-8 (clever, eh?).

### *UTF-8 Example*

Hello-UTF-8:

```
EF BB BF 48 65 6C 6C 6F
header   H  e  l  l  o
```

Again, the ASCII text is not changed in UTF-8. Feel free to use charmap to copy in some Unicode characters and see how they are stored in UTF-8. Or, you can experiment online.

### *UTF-7*

While UTF-8 is great for ASCII, it still stores Unicode data as non-ASCII characters with the high-bit set. Some email protocols do not allow non-ASCII values, so UTF-8 data would not be sent properly. Systems that can handle data with anything in the high bit are “8-bit clean”; systems that require data have values 0-127 (like SMTP) are not. So how do we send Unicode data through them?

Enter UTF-7. The goal is to encode Unicode data in 7 bits (0-127), which is compatible with ASCII. UTF-7 works like this

- Codepoints in the ASCII range are stored as ASCII, except for certain symbols (+, -) that have special meaning
- Codepoints above ASCII are converted to binary, and stored in base64 encoding (stores binary information in ASCII)

How do you know which ASCII letters are real ASCII, and which are base64 encoded? Easy. ASCII characters between the special symbols “+” and “-“ are considered base64 encoded.

“-” acts like an escape suffix character. If it follows a character, that item is interpreted literally. So, “+-“ is interpreted as “+” without any special encoding. This is how you store an actual “+” symbol in UTF-7.

### *UTF-7 Example*

Wikipedia has some UTF-7 examples, as Notepad can’t save as UTF-7.

“Hello” is the same as ASCII — we are using all ASCII characters and no special symbols:

```
Byte:     48 65 6C 6C 6F
Letter:   H  e  l  l  o
```

“£1” (1 British pound) becomes:

```
+AKM-1
```

The characters “+AKM-” means AKM should be decoded in base64 and converted to a codepoint, which maps to 0x00A3 or the British pound symbol. The “1” is kept the same, since it is a ASCII character.

UTF is pretty clever, eh? It’s essentially a Unicode to ASCII conversion that removes any characters that have their highest-bit set. Most ASCII characters

will look the same, except for the special characters (- and +) that need to be escaped.

### *Wrapping it up – what I've learned*

I'm still a newbie but have learned a few things about Unicode:

- Unicode does not mean 2 bytes. Unicode defines code points that can be stored in many different ways (UCS-2, UTF-8, UTF-7, etc.). Encodings vary in simplicity and efficiency.
- Unicode has more than 65,535 (16 bits) worth of characters. Encodings can specify more characters, but the first 65535 cover most of the common languages.
- You need to know the encoding to correctly read a file. You can often *guess*that a file is Unicode based on the Byte Order Mark (BOM), but confusion can still arise unless you know the exact encoding. Even text that looks like ASCII could actually be encoded with UTF-7; you just don't know.

Unicode is an interesting study. It opened my eyes to design tradeoffs, and the importance of separating the core idea from the encoding used to save it