# Unicode and Character Sets

Content-Type in html

1. Characters
2. Character Sets
3. Unicode
4. Encoding

# ASCII

32 to 127 represent every character.

32 – Space

65 – A

This requires 7 bits to store characters.

Most computers were using 8-bit bytes, so you had a whole bit to spare.

Codes below 32 were called unprintable and they were used for control characters.

128 – 255 – lots of people had this idea what should go in here. OEM Characters

# Code Pages

For example, on some PCs the character code 130 would display e, but on computers sold in Israel it was the Hebrew letter #. So when Americans would send their resumes to Israel, it would arribe as R#sum#.

Code Pages: What do 128 and above represent for different countries.

# Unicode

In Unicode, a letter maps to something called a code point which is still just a theoretical concept.

Every platonic letter in every alphabet is assigned a magic number by the Unicode consortium which is written like this: U+0639. This magic number is called a code point.

The U+ means “Unicode” and the numbers are hexadecimal.

U+0639 is the Arabic letter Ain.

U+0041 – letter A.

There is no real limit on the number of letters that Unicode can define and infact they have gone beyond 65536, so not every Unicode letter can really be squeezed into two bytes.

String: “Hello”

Unicode code points – U+0048 U+0065 U+006C U+006C U+006F

How to store this in memory or represent it in an email message?

# Encodings

That’s where encodings come in.

Earliest idea for Unicode encoding – Let’s just store those numbers in two bytes each.

So “Hello” becomes

00 48 00 65 00 6C 00 6C 00 6F

It could also be

48 00 65 00 6C 00 6C 00 6F 00

It could be high-endian or low-endian mode, whichever their particular CPU was fastest at.

To solve this store FE FF at the beginning of every Unicode string; this is called a Unicode Byte Order Mark and if you are swapping your high and low bytes it will look like a FF FE and the person reading your string will know that they have to swap every other byte.

Two bytes for storing Unicode characters was too much for English letters which hardly goes above U+00FF.

# Encodings – UTF-8

UTF-8 stores your string of Unicode code points (magic U+ numbers) in memory using 8 bit bytes.

In UTF-8, every code point from 0-127 is stored in a single byte. Only code points 128 and above are stored using 2, 3, in fact, up to 6 bytes.

This has the neat side effect that English text looks exactly the same in UTF-8 as it did in ASCII, so Americans don’t even notice anything wrong.

Only the rest of the world has to jump through hoops.

Specifically Hello which was

U+0048 U+0065 U+006C U+006C U+006F will be stored as 48 65 6C 6C 6F which is same as it was stored in ASCII and ANSI and every OEM character set on the planet.

So far, we have covered two ways of encoding:

1. UCS-2 (it has two bytes) or UTF-16
2. UTF-8

There are a bunch of other ways of encoding Unicode.

UTF-7,

UCS-4 which stores each code point in 4 Bytes

And in fact now that you’re thinking of things in terms of platonic ideal letters

which are represented by Unicode code points, those unicode code points can

be encoded in any old-school encoding scheme, too! For example, you could

encode the Unicode string for Hello (U+0048 U+0065 U+006C U+006C U+006F) in

ASCII, or the old OEM Greek Encoding, or the Hebrew ANSI Encoding, or any of

several hundred encodings that have been invented so far, with one catch: some

of the letters might not show up! If there’s no equivalent for the Unicode code

point you’re trying to represent in the encoding you’re trying to represent it in,

you usually get a little question mark: ? or, if you’re really good, a box.

# Important fact about encodings

It does not make sense to have a string without know what encoding it uses.

How do we preserve this information about what encoding a string uses?

For an email message, you are expected to have a string in the header of the form

Content-Type: text/plain; charset=”UTF-8”

Content-Type of the HTML file is put right in the HTML file itself.

<html>

<head>

<meta http-equiv="Content-Type" content="text/html; charset=utf-8">

But that meta tag really has to be the very first thing in the <head> section because as soon as the web browser sees this tag it’s going to stop parsing the page and start over after reinterpreting the whole page using the encoding you specified.

# What do web browsers do if they don’t find any Content-Type either in HTTP headers or the meta tag?

Internet Explorer actually does something quite interesting: it tries to guess, based on the frequency in which various bytes appear in typical text in typical encodings of various languages, what language

and encoding was used. Because the various old 8 bit code pages tended to put their national letters in different ranges between 128 and 255, and because every human language has a different characteristic histogram of letter usage, this actually has a chance of working. It’s truly weird, but it does seem to work

often enough that naïve web-page writers who never knew they needed a Content-Type header look at their page in a web browser and it looks ok, until one day, they write something that doesn’t exactly conform to the letter-frequency-distribution of their native language, and Internet Explorer decides

it’s Korean.

specified.