

## Re: C# and encodings

---

*Source:*

<http://www.tech-archive.net/Archive/DotNet/microsoft.public.dotnet.languages.csharp/2009-02/msg00280.html>

---

- *From:* "Peter Duniho" <NpOeStPeAdM@xxxxxxxxxxxxxxxxxxxx>
  - *Date:* Tue, 03 Feb 2009 15:41:06 -0800
- 

On Tue, 03 Feb 2009 15:08:59 -0800, <beginwithl@xxxxxxxxxx> wrote:

1)

a) With "Encoding.Default" you retrieve system s default code page. But if windows has numerous code pages, then what exactly would default page be, meaning where ( or in what apps ) does windows use this default page over other code pages?

Windows only has one current code page at a time.

b) Can code page support Unicode coded character set, but may use different encoding than Unicode does ( Unicode set uses three encodings – UTF-8, UTF-16 and UTF-32 )?

The code page can be and often is not Unicode. Any character encoding that is not Unicode by definition uses a different encoding than Unicode does.

c)

\* Are there also 8-bit code pages which use Unicode character encoding, and thus have only 255 code points matched to characters?

That makes no sense. Unicode can't be represented in only 8-bits, so there's no such thing as an 8-bit code page that uses Unicode character encoding.

\* Can these code pages also use UTF-16 or UTF-32 encoding?

UTF-16 and UTF-32 are also Unicode. See above.

## Re: C# and encodings

\* Are there also code pages that support more than 255, but less than  $2^{16}$  code points?

Certainly.

2)

From MSDN site:

StreamWriter defaults to using an instance of UTF8Encoding unless specified otherwise. This instance of UTF8Encoding is constructed such that the Encoding.GetPreamble method returns the Unicode byte order mark written in UTF-8. The preamble of the encoding is added to a stream when you are not appending to an existing stream. This means any text file you create with StreamWriter will have three byte order marks at its beginning."

As far as I understand, the above text suggests that preamble should be added by default, but I'd say that's not true?!

You are welcome to say that.

3) I noticed there are only four classes derived from Encoding class ( ASCIIEncoding, UTF8Encoding, UnicodeEncoding and UTF7Encoding ). What if you want to use some other, non-unicode encoding?

You use them instead. See Encoding.GetEncoding().

4)

a)

From MSDN site:

StreamWriter Constructor (Stream, Encoding)

If you specify something other than Encoding.Default, the byte order mark (BOM) is written to the file.

But BOM should only be added when using one of Unicode encodings, thus why would BOM be added if you specify non-Unicode encoding?

Three possibilities:

- the documentation is wrong
- the implementation is wrong
- your assumption about when the BOM should be added is wrong

## Re: C# and encodings

I don't have enough first-hand knowledge to choose among those three at the moment.

b) Since the Unicode byte order mark character is not found in any code page, it disappears if data is converted to ANSI. Unlike other Unicode characters, it is not replaced by a default character when it is converted. If a byte order mark is found in the middle of a file, it is not interpreted as a Unicode character and has no effect on text output.

Well, since at least some (ANSI) code pages do have glyphs for characters at code points FF and FE, I assume above text implies that apps ( using non-unicode code pages ) reading such a file would understand that FE FF sequence represents BOM and thus should ignore it?

I believe the statement refers to converting from Unicode to an ANSI code page, not the other way. The point is not whether 0xff or 0xfe can be found on their own. The point is that the Unicode "character" 0xfeff is not representable in any ANSI code page, and is treated specially by stripping it from input rather than replacing it with the "default character".

In other words, it is up to app ( using non-Unicode code page ) reading such a file to realize that FE FF sequence should be ignored?!

If you use Encoding to convert to an ANSI code page, it will be ignored automatically.

I don't really even know when this would show up in practice. I suppose if you were writing out a BOM character explicitly in a text-based (string or char) API that then got converted to an ANSI code page, it might come up there.

5)  
a) Internally, the .NET Framework stores text as Unicode UTF-16.

I assume that the above quote is only referring to String objects and char variables using UTF-16 encoding, or is there some other text which is also stored as UTF-16?

As far as I know, string and char are it. I can't imagine the point of having some other, identically functional data structure.

b) Ignoring the fact that FE FF sequence identifies the type of encoding, does U+FEFF also represent a character ( outside the context of encoding )?

## Re: C# and encodings

It's the Unicode BOM character. It's an actual Unicode character.

6)  
Say app1 ( running on PC1 ) and app2 ( running on PC2 ) communicate via network using TCP/IP protocol. PC1 uses little endian-order, while PC2 uses big-endian order. Now, I know we send information over TCP/IP ( and networks in general ) using big-endian order,

We do?

I send information over TCP/IP in whatever byte order seems appropriate at the time. TCP/IP just uses bytes in whatever order you provide them. This can be big-endian or little-endian. Whatever you want.

but:

a) But does only data in the packet s header uses this byte order, while application data is sent just as it is, without reversing its byte order ( assuming this data is sent over the network by PC1 )?

What packet? An IP datagram? An application message?

IP uses a specific format for its internal data structures, but you do not need to worry about that at all unless you're writing a network driver, hardware firmware, etc.

The application data is sent exactly as it's provided by the application, and is received exactly the same way. The network i/o is completely agnostic to byte ordering.

b) If so, then if PC1 sends some .exe file to PC2, then how will PC2 know whether it came from little endian-machine and thus should reverse bytes before trying to load this .exe file?

It would be unusual for a computer that has big-endian hardware to be able to run an executable designed for a computer with little-endian hardware. Most CPUs aren't configurable; they are either always little-endian, or always big-endian.

I don't even recall off the top of my head the examples of CPUs that are configurable, but my recollection is that in those very rare situations, the architecture is designed to handle either byte-order for executables.

In other words, there's never a situation where "PC2...should reverse bytes before trying to load this .exe file".

Pete

.