The Notepad file encoding problem, redux

devblogs.microsoft.com/oldnewthing/20070417-00

April 17, 2007

Raymond Chen

About every ten months, somebody new discovers the Notepad file encoding problem. Let's see what else there is to say about it.

First of all, can we change Notepad's detection algorithm? The problem is that there are a lot of different text files out there. Let's look just at the ones that Notepad supports.

- 8-bit ANSI (of which 7-bit ASCII is a subset). These have no BOM; they just dive right in with bytes of text. They are also probably the most common type of text file.
- UTF-8. These usually begin with a BOM but not always.
- Unicode big-endian (UTF-16BE). These usually begin with a BOM but not always.
- Unicode little-endian (UTF-16LE). These usually begin with a BOM but not always.

If a BOM is found, then life is easy, since the BOM tells you what encoding the file uses. The problem is when there is no BOM. Now you have to guess, and when you guess, you can guess wrong. For example, consider this file:

```
D0 AE
```

Depending on which encoding you assume, you get very different results.

- If you assume 8-bit ANSI (with code page 1252), then the file consists of the two characters `U+00D0 U+00AE`, or "Ð®". Sure this looks strange, but maybe it's part of the word VATNIÐ® which might be the name of an Icelandic hotel.
- If you assume UTF-8, then the file consists of the single Cyrillic character `U+042E`, or "Ю".
- If you assume Unicode big-endian, then the file consists of the Korean Hangul syllable `U+D0AE`, or "킮".
- If you assume Unicode little-endian, then the file consists of the Korean Hangul syllable `U+AED0`, or "껐".

Okay, so this file can be interpreted in four different ways. Are you going to use the "try to guess" algorithm from `IsTextUnicode`? (Michael Kaplan has some thoughts on this subject.) If so, then you are right where Notepad is today. Notice that all four interpretations

are linguistically plausible.

Some people might say that the rule should be "All files without a BOM are 8-bit ANSI." In that case, you're going to misinterpret all the files that use UTF-8 or UTF-16 and don't have a BOM. Note that the Unicode standard even advises **against** using a BOM for UTF-8, so you're already throwing out everybody who follows the recommendation.

Okay, given that the Unicode folks recommend against using a BOM for UTF-8, maybe your rule is "All files without a BOM are UTF-8." Well, that messes up all 8-bit ANSI files that use characters above 127.

Maybe you're willing to accept that ambiguity, and use the rule, "If the file looks like valid UTF-8, then use UTF-8; otherwise use 8-bit ANSI, but under no circumstances should you treat the file as UTF-16LE or UTF-16BE." In other words, "never auto-detect UTF-16". First, you still have ambiguous cases, like the file above, which could be either 8-bit ANSI or UTF-8. And second, you are going to be flat-out wrong when you run into a Unicode file that lacks a BOM, since you're going to misinterpret it as either UTF-8 or (more likely) 8-bit ANSI. You might decide that programs that generate UTF-16 files without a BOM are broken, but that doesn't mean that they don't exist. For example,

```
cmd /u /c dir >results.txt
```

This generates a UTF-16LE file without a BOM. If you poke around your Windows directory, you'll probably find other Unicode files without a BOM. (For example, I found `COM+.log`.) These files still "worked" under the old `IsTextUnicode` algorithm, but now they are unreadable. Maybe you consider that an acceptable loss.

The point is that no matter how you decide to resolve the ambiguity, somebody will win and somebody else will lose. And then people can start experimenting with the "losers" to find one that makes your algorithm look stupid for choosing "incorrectly".

Raymond Chen

Follow