JOEL ON SOFTWARE

I’m Joel Spolsky, a software developer in New York City. More about me.

OCTOBER 8, 2003 *by* JOEL SPOLSKY

The Absolute Minimum Every Software Developer Absolutely, Positively Must Know About Unicode and Character Sets (No Excuses!)

TOP 10, NEW DEVELOPER, NEWS

Ever wonder about that mysterious Content-Type tag? You know, the one you're supposed to put in HTML and you never quite know what it should be?

Did you ever get an email from your friends in Bulgaria with the subject line "???? ?????? ??? ????"?

I've been dismayed to discover just how many software developers aren't really completely up to speed on the mysterious world of character sets, encodings, Unicode, all that stuff. A couple of years ago, a beta tester for FogBUGZ was wondering whether it could handle incoming email in Japanese. Japanese? They have email in Japanese? I had no idea. When I looked closely at the commercial ActiveX control we were using to parse MIME email messages, we discovered it was doing exactly the wrong thing with character sets, so we actually had to write heroic code to undo the wrong conversion it had done and redo it correctly. When I looked into another commercial library, it, too, had a completely broken character code implementation. I corresponded with the developer of that package and he sort of thought they "couldn't do anything about it." Like many programmers, he just wished it would all blow over somehow.

So I have an announcement to make: if you are a programmer working in 2003 and you don't know the basics of characters, character sets, encodings, and Unicode, and I *catch* you, I'm going to punish you by making you peel onions for 6 months in a submarine. I swear I will.

And one more thing:

IT'S NOT THAT HARD.

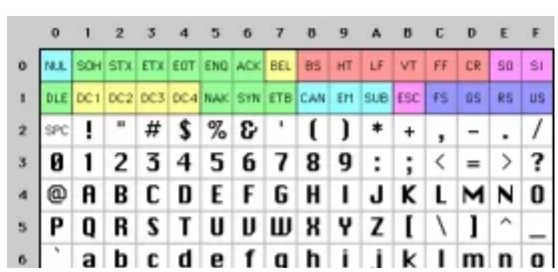
In this article I'll fill you in on exactly what *every working programmer* should know. All that stuff about "plain text = ascii = characters are 8 bits" is not only wrong, it's hopelessly wrong, and if you're still programming that way, you're not much better than a medical doctor who doesn't believe in germs. Please do not write another line of code until you finish reading this article.

Before I get started, I should warn you that if you are one of those rare people who knows about internationalization, you are going to find my entire discussion a little bit oversimplified. I'm really just trying to set a minimum bar here so that everyone can understand what's going on and can write code that has a *hope* of working with text in any language other than the subset of English that doesn't include words with accents. And I should warn you that character handling is only a tiny portion of what it takes to create software that works internationally, but I can only write about one thing at a time so today it's character sets.

A Historical Perspective

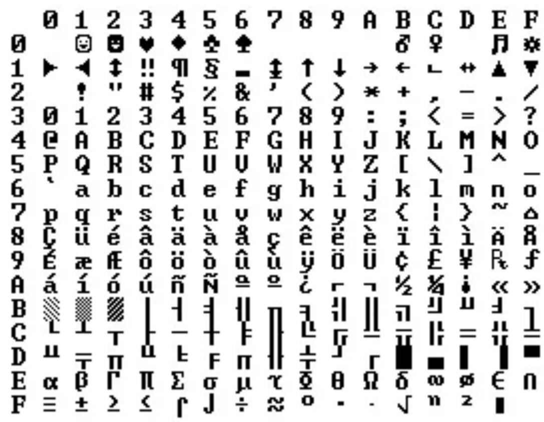
The easiest way to understand this stuff is to go chronologically.

You probably think I'm going to talk about very old character sets like EBCDIC here. Well, I won't. EBCDIC is not relevant to your life. We don't have to go that far back in time.



days were using 8-bit bytes, so not only could you store every possible ASCII character, but you had a whole bit to spare, which, if you were wicked, you could use for your own devious purposes: the dim bulbs at WordStar actually turned on the high bit to indicate the last letter in a word, condemning WordStar to English text only. Codes below 32 were called *unprintable* and were used for cussing. Just kidding. They were used for control characters, like 7 which made your computer beep and 12 which caused the current page of paper to go flying out of the printer and a new one to be fed in.

And all was good, assuming you were an English speaker.



Because bytes have room for up to eight bits, lots of people got to thinking, “gosh, we can use the codes 128-255 for our own purposes.” The trouble was, *lots* of people had this idea at the same time, and they had their own ideas of what should go where in the space from 128 to 255. The IBM-PC had something that came to be known as the OEM character set which provided some accented characters for European languages and a bunch of line drawing characters… horizontal bars, vertical bars, horizontal bars with little dingle-dangles dangling off the right side, etc., and you could use these line drawing characters to make spiffy boxes and lines on the screen, which you can still see running on the 8088 computer at your dry cleaners’. In fact as soon as people started buying PCs outside of America all kinds of different OEM character sets were dreamed up, which all used the top 128 characters for their own purposes. For example on some PCs the character code 130 would display as é, but on computers sold in Israel it was

Icelandic and they even had a few “multilingual” code pages that could do Esperanto and Galician *on the same computer! Wow!* But getting, say, Hebrew and Greek on the same computer was a complete impossibility unless you wrote your own custom program that displayed everything using bitmapped graphics, because Hebrew and Greek required different code pages with different interpretations of the high numbers.

Meanwhile, in Asia, even more crazy things were going on to take into account the fact that Asian alphabets have thousands of letters, which were never going to fit into 8 bits. This was usually solved by the messy system called DBCS, the “double byte character set” in which *some* letters were stored in one byte and others took two. It was easy to move forward in a string, but dang near impossible to move backwards. Programmers were encouraged not to use s++ and s– to move backwards and forwards, but instead to call functions such as Windows’ AnsiNext and AnsiPrev which knew how to deal with the whole mess.

But still, most people just pretended that a byte was a character and a character was 8 bits and as long as you never moved a string from one computer to another, or spoke more than one language, it would sort of always work. But of course, as soon as the Internet happened, it became quite commonplace to move strings from one computer to another, and the whole mess came tumbling down. Luckily, Unicode had been invented.

Unicode

In Unicode, the letter A is a platonic ideal. It's just floating in heaven:

A

This platonic A is different than B, and different from a, but the same as A and ***A*** and A. The idea that A in a Times New Roman font is the same character as the A in a Helvetica font, but *different* from "a" in lower case, does not seem very controversial, but in some languages just figuring out what a letter *is* can cause controversy. Is the German letter ß a real letter or just a fancy way of writing ss? If a letter's shape changes at the end of the word, is that a different letter? Hebrew says yes, Arabic says no. Anyway, the smart people at the Unicode consortium have been figuring this out for the last decade or so, accompanied by a great deal of highly political debate, and you don't have to worry about it. They've figured it all out already.

Every platonic letter in every alphabet is assigned a magic number by the

The earliest idea for Unicode encoding, which led to the myth about the two bytes, was, hey, let's just store those numbers in two bytes each. So Hello becomes

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

Right? Not so fast! Couldn't it also be:

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

Well, technically, yes, I do believe it could, and, in fact, early implementors wanted to be able to store their Unicode code points in high-endian or low-endian mode, whichever their particular CPU was fastest at, and lo, it was evening and it was morning and there were already *two* ways to store Unicode. So the people were forced to come up with the bizarre convention of storing a

00200000	03FFFFFF	111110vv	10vvvvvv	10vvvvvv	10vvvvvv	10vvvvvv	
04000000	7FFFFFFF	1111110v	10vvvvvv	10vvvvvv	10vvvvvv	10vvvvvv	10vvvvvv

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, behold! is the same as it was stored in ASCII, and ANSI, and every OEM character set on the planet. Now, if you are so bold as to use accented letters or Greek letters or Klingon letters, you'll have to use several bytes to store a single code point, but the Americans will never notice. (UTF-8 also has the nice property that ignorant points correctly and change all the other code points into question marks. Some popular encodings of English text are Windows-1252 (the Windows 9x standard for Western European languages) and ISO-8859-1, aka Latin-1 (also useful for any Western European language). But try to store Russian or Hebrew letters in these encodings and you get a bunch of question marks. UTF 7, 8, 16, and 32 all