A brief and also incomplete history of Windows localization

devblogs.microsoft.com/oldnewthing/20120726-00

July 26, 2012

Raymond Chen

The process by which Windows has been localized has changed over the years. Back in the days of 16-bit Windows, Windows was developed with a single target language: English. Just English. After Windows was complete and masters were sent off to the factory for duplication, the development team handed the source code over to the localization teams. “Hey, by the way, we shipped a new version of Windows. Go localize it, will ya?” While the code that was written for the English version was careful to put localizable content in resources, there were often English-specific assumptions hard-coded into the source code. For example, it may have assumed that the text reading direction was left-to-right or assumed that a single character fit in a single byte. (Unicode hadn’t been invented yet.) The first assumption is not true for languages such as Hebrew and Arabic (which read right-to-left), and to a lesser degree Chinese and Japanese (which read top-to-bottom in certain contexts). The second assumption is not true for languages like Chinese, Japanese, and Korean, which use DBCS (double-byte character sets). The localization teams made the necessary code changes to make Windows work in these other locales and merged them back into the master code base. The result was that there were three different versions of the *code* for Windows, commonly known as Western, Middle-East, and Far-East. If you wanted Windows to support Chinese, you had to buy the Far-East version of Windows. And since the code was different for the three versions, they had different sets of bugs, and workarounds for one version didn’t always work on the others. (Patches didn’t exist back then, there being no mechanism for distributing them.) If you ran into a problem with a Western language, like say, German, then you were out of luck, since there was no German Windows code base; it used the same Western code base. Windows 95 tried out a crazy idea: Translate Windows into German *during the development cycle*, to help catch these only-on-German problems while there was still time to do something about it. This, of course, created significant additional expense, since you had to have translators available throughout the product cycle instead of hiring them just once at the end. I remember catching a few translation errors during Windows 95: A menu item *Sort* was translated as *Art* (as in “What sort of person would do this?”) rather than *Sortieren* (“put in a prearranged order”). And a command line tool asked the user a yes/no question, promting “J/N” (*Ja/Nein*), but if you wanted to answer in the affirmative, you had to type “Y”. The short version of the answer to the question “Why can't the localizers change the code if they have to?" is "Because the code already shipped. What are you going to do, recall every copy of Windows?" At least in Windows 95, the prohibition on changing code was violated if circumstances truly demanded them, but doing so was very expensive. The only one I can think of is the change to remove the time zone highlighting from the world map. And the change was done in the least intrusive possible way: Patching four bytes in the binary to make the highlight and not-highlight colors the same. You dare not do something like introduce a new variable; who knows what kinds of havoc could result! Having all these different versions of Windows made servicing very difficult, because you had to develop and test a different patch for each code base. Over the years, the Windows team has developed techniques for identifying these potential localization problems earlier in the development cycle. For a time, Windows was "early-localized" into German and Japanese, so as to cover the Western and Far-East scenarios. Arabic was added later, expanding coverage to the Mid-East cases, and Hindi was added in Windows 7 to cover languages which are Unicode-only. Translating each internal build of Windows has its pros and cons: The advantage is that it can find issues when there is still time to make code changes to address them. The disadvantage is that code can change while you are localizing, and those code changes can invalidate the work you've done so far, or render it pointless. For example, somebody might edit a dialog you already spent time translating, forcing you to go back and re-translate it, or at least verify that the old translation still works. Somebody might take a string that you translated and start using it in a new way. Unless they let you know about the new purpose, you won't know that the translation needs to be re-evaluated and possibly revised. The localization folks came up with a clever solution which gets most of the benefits while avoiding most of the drawbacks: They invented pseudo-localization, which simulates what Michael Kaplan calls "an eager and hardworking yet naïve intern localizer who is going to translate every single string." This was so successful that they hired a few more naïve intern localizers, one which performed "Mirrored pseudo-localization" (covering languages which read right-to-left) and "East Asian pseudo-localization" (covering Chinese, Japanese, and Korean).

But the rule prohibiting code changes remains in effect. Changing any code resets escrow, which means that the ship countdown clock gets reset back to its original value and all the testing performed up until that point needs to be redone in order to verify that the change did not affect them.

Raymond Chen

Follow