

Your Passport to Proper Internationalization

From Multibyte Coding System character set nightmares to tricky Western SQL database support, from GUIs that assume Latin font metrics to poorly planned parallel development, myriad mishaps can occur in what looks like a simple localization exercise.

by Benson I. Margulies

You've probably heard horror stories about how hard it is to modify software to work in Japanese or other Asian languages. On the other hand, perhaps you've encountered claims that such a process is a simple matter of extracting strings and translating them. In fact, there are many potential procedural and technical pitfalls in building international software, and there is no substitute for thinking carefully about the problem and formulating a plan of attack.

How Not To Internationalize

Here's a cautionary tale about a company—we'll call it Acme Products—that did not take an organized approach to internationalization.

The trouble began when the company decided to enter the Asian marketplace. Acme hired a new vice president for Asia, Clive, who in turn hired a new sales and marketing staff. Clive's team members spent all their time and energy firming up distribution relationships in Japan. They made no attempt to verify that the engineering was being done to provide them a product to deliver in Japan. It wasn't long before they had closed the first big deal.

Clive arrived at the corporate office about a month before the next major release of Acme's product was due to be deployed—just after feature freeze. Not only was making changes to the new release unthinkable, but all of the developers were already working flat-out on the release. They had no time to discuss a Japanese version of the previous release, let alone build one. Clive had no choice but

to bring in new people to build the Japanese version. With no access to the over-occupied original architects or developers, they used the previous release as a black box in their attempt to fulfill their mandate: "Make it work in Japan very quickly."

Clive wasn't a technical person. As far as he knew, this was a "localization" problem—all the English strings in the product needed to be in Japanese. No big deal.

Given that direction, the team set out to do a strict localization: Find the strings and translate them, and nothing else. They didn't anticipate what, if any, problems the code might exhibit when faced with Japanese text, and they didn't consider any new features.

Acme's development manager, Sam, didn't want to hear about this project. It had come out of nowhere, with a budget that was not under his control. He told his people to make a source snapshot of the product and "throw it over the wall" to the "outsiders" on the Japanese project. Sam's job was done.

Acme's programming shop was not accustomed to parallel software development on any scale. The release manager, Rhonda, was in charge of source control in conjunction with her release group. They used a source-management system with a single line of development from release to release and handled patches somewhat informally. They weren't prepared to handle an outside team making significant changes to the code in parallel with their ongoing development and maintenance process. To set up a real parallel development branch would

have been quite costly the first time—especially since the Japanese project was off-site. Since this was neither in the budget nor the plan, no one in Rhonda’s group wanted to hear about it. Rhonda ignored the Japanese project.

The intrepid Japanese development team got down to work anyway. Their main problem was strings—or so they thought. They had heard somewhere that message catalogs or resources were the “right” way to handle these, but catalogs or resources looked like too much work for the time they had allocated. The code was in C and had constructs such as:

```
static char *strx[] = "A typical string:  
our Hovercraft is Full of Eels";
```

The compiler wouldn’t let them replace that string with a function call. Other strings were in performance-sensitive spots where the cost of a function call would be deadly. They simply didn’t have time to deal with the changes required, so they made a fateful decision: They built scripts to replace the strings in-line, creating a mutant version of the code base with all the English replaced by Japanese.

In order to put Japanese strings into the source, they had to get some Japanese strings. That wasn’t too hard. Plenty of folks “know Japanese.” The team didn’t consider whether those people were qualified to translate technical terms. After all, they were in a hurry, and one of the team member’s boyfriends could knock off the work in the evenings in no time flat.

Once the translation was done, the developers stuck it into the code without further thought. No one edited it.

The Acme Japanese development team started having problems with testing. To begin with, Acme had few, if any, formalized testing procedures. There was a quality assurance team, and the members of that team had a body of folk wisdom for their release testing. There were some automated scripts, but these all assumed English text in the user interface. Plus, the scripts were implemented in a testing tool that didn’t support Japanese. The team struggled with the difficulty of obtaining, installing and maintaining localized Japanese versions of the operating systems for testing.

Finally, they delivered the first beta version to Japan. Soon after, the defect reports began to arrive. The team had expected some minor visual blemishes and idiomatic inaccuracies. They hadn’t expected the following:

Storage corruption. Pointers and other data were corrupted due to buffer overflow.

Ungainly dialog boxes. Acme’s product had a Windows GUI, which, like many GUIs, had a population of dialog boxes. Some were carefully designed to fit into 800x600 pixels. Imagine the team’s surprise when they discovered that the dialog boxes didn’t fit into 800x600 on Japanese Windows, even without any Japanese text!

Mysterious database errors. The application worked with a SQL database. Down in the depths of the product, the code set is specified by an obscure parameter that determined the character encoding for communications with the database. In this product, the parameter depended on a default setting that didn’t support Japanese.

The Japanese team had little database expertise, and Acme’s database experts were far too busy with the new release to be bothered. Not only didn’t the team know about the environmental parameter that controlled the character encoding used on the client, they didn’t know that there was a critical parameter that had to be set when the database was installed and configured. As a result, they went along using a “Western European” database and could not understand the strange misbehaviors and errors that resulted when they tried to store Japanese text in some fields. Furthermore, they just didn’t understand why fields that were always long enough in English weren’t long enough in Japanese.

Poor translation. The translators weren’t qualified, and no one had checked their work.

Missing features. The product collected first and last names, but the developers had not made provisions for the pronounceable versions of Japanese names.

Against the odds, the team eventually delivered a product. It was late, it was missing critical features, and it had bugs. (Other than that, it was fine.) When mediocre sales followed, the top managers all blamed each other. Clive, the Asia vice president, blamed Sam and Rhonda in development and release management for failing to support his effort, while Sam and Rhonda felt that the mediocre sales proved what they had been saying all along—there is no real money to be made in Asia.

The Japanese development team tried to merge their work into the source for the next release, but without a code branch in a source management tree (a basic capability of a good source management tool), this was impossible. In any case, the in-line

Japanese strings couldn't be checked in replacing the English strings, and the Japanese version was now in permanent limbo. Japanese customers who encountered defects had to wait for someone to manually port the fix from the main English code base to the Japanese version.

In spite of the difficulties, Clive did not give up. He doggedly scoured for sales opportunities. And he found them. However, he had a huge problem—the new customers required features from the new release. And some of them were in China instead of Japan. Once again, the weary team suited up, resigned to starting the entire process over again. And here we leave them, slogging along and cursing their fate, the perpetual outsiders. The developers hate them, and they aren't too popular with Clive either because of the product's poor performance. Who would want to work on a team like this one?

What Went Wrong?

The Acme project suffered from a series of problems, and each problem fed the next. That doesn't mean that you can't get into trouble by only following parts of their example; you can get into big trouble with any one of them. To avoid internationalization problems, consider your organizational structure, process and technical requirements. Here are some of the specific mistakes in the Acme story:

Bad organizational structure. You can't get a good result from internationalization if you don't treat it as part of your core business. If you ghettoize your international initiatives, they will suffer from the lack of communication, coordination and buy-in. It's a good idea to have a separate marketing and sales vice president focused on the specific regional and cultural issues.

Failure to coordinate. Because the internationalization process wasn't integrated into the overall development plan, the Asian requirements—which merited the initial attention of the architects and developers—weren't included in the release.

Failure to understand the requirements. As with many other foreign markets, Japan has unique needs. For example, Acme's product required support for pronounceable Japanese strings, called furigana. These must, in some cases, be displayed atop a string in a typesetting convention called ruby text. The product also needed support for the Input Method Editor that allows the input of ideographic

characters, as well as support for the Japanese rules for breaking lines. Since no one took the time to analyze these potential requirements, they were all left out. The first mistake was disastrous: Without furigana, Acme's customer service representatives couldn't pronounce the customers' names.

Failure to appreciate the technical challenges. Even taking English software to Europe can yield snags. Acme had a much harder problem in aiming for Japan. Here are a few of the biggest obstacles they slammed into:

- Their GUIs embodied English grammar and needed significant rearrangement.
- The GUIs assumed Latin font metrics and thus didn't fit on the screen in Asia.
- In Asian languages, text is represented in the Multibyte Coding System (MBCS). Acme's code used some of the common clichés for strings that corrupt MBCS text. See the sidebar for some gory details.
- Acme's product used a database and foundered on the complexities of the SQL database's international support.
- The code sorted strings by pure numerical order.
- The code used third-party components that didn't work for international text.

The code included latent defects, especially storage defects, that are common to C or C++ code. Without excellent test coverage or a good storage defect detection product, these problems lie low. Heaps are surprisingly tolerant of some mistakes, but changing the lengths of strings is a good way to change the allocation pattern and convert latent defects into blatant defects.

In short, there is no such thing as "localizing" a body of code that has never been internationalized before. This is always an internationalization project. It might be a small one or a large one, but it is never absent. Once the code has been internationalized and the changes integrated into the code base, then there is the possibility of pure localization to adapt it to additional countries.

Failure to account for parallel development. The team made no allowances for proper source management during parallel development. Internationalization projects are textbook examples of the importance of source management. They involve many small changes to a large body of code,

and they must often occur simultaneously with other development efforts.

Weak translation and editing. Acme is something of an extreme case. Many companies setting out to localize at least manage to hire a professional translator. However, even a professional translator does not guarantee success. What was missing from this picture? Editing. To speak in bald economic terms, translators are often paid by the word. They have an incentive to rush. And, like the rest of us, they make mistakes. Translators are a species of writer, and any good writer knows the adage, “He who proofs his own copy has a fool for an editor.”

Leaving strings in place. The quickest way to localize code is to take the source and replace the English strings with strings in another language. It may be quick, but it essentially dooms any attempt to maintain a single source that works in multiple countries.

Testing. Acme’s problems with testing were like Acme’s problems with source management. A relatively informal development process that worked “well enough” for one country didn’t work well at all for a multilingual product. Acme wasn’t prepared to have international testers show up and do productive work.

A Spoonful of Process

To succeed with internationalization, modify your development process to accommodate the effort. You’ll find that internationalization turns up at all phases of the project. This may sound daunting, but keep in mind that a small amount of effort and attention early on will save you a lot of work down the line.

Successful internationalization starts with good communication among all the participants. If there is a specific international sales and marketing team, its management should establish strong lines of communication to the development management team. The international team has to understand the development process, including the flow of requirements into design and the schedule for releases. A successful international initiative must be launched in the entire company, not just in an international group, subsidiary or division. It has to be sold to everyone, and the budget has to make realistic provisions for everyone’s efforts.

Ideally, internationalization should be the responsibility of the core development group. They know the code best. If international support is a full-fledged requirement for a normal release, then the developers can ensure that the required changes are integrated into the architecture. This is an important part of giving the entire enterprise an international outlook. Even if the core developers can’t do the work, it is a good idea for their managers to have some oversight over the internationalization developers.

If, for reasons of budget or schedule, internationalization has to live outside of the core development organization, it is especially important to foster strong communications between the international sales and marketing team, the international development team and the core development team.

Requirements

If you want your code to support international deployment, say so at the very beginning. It is much easier to build U.S.-only code than international code. Schedule-pressed developers will never allow for international support if you don’t make it part of the requirements. Don’t listen if someone tells you that international support is zero-cost if you only take it into account at the outset. That’s not true. It will cost something. Keeping it in mind from the outset, however, will make it cost much less.

Internal requirements take two major forms: the language-neutral requirements and the specific target requirements. Examples of language-neutral requirements are:

- The code shall support localization with few or no modifications to executable code. This includes dialog layouts.
- The code shall operate properly with MBCS international text. (Or with Unicode, another way of dealing with MBCS languages).
- The code shall operate correctly with text in several different languages.
- The code shall use local-sensitive sorting, currency formatting and date formatting.

The second flavor is specific requirements for specific target markets. Here are some examples:

- Japanese furigana, as previously described.
- Chinese elaborate numbers. In addition to Latin digits, the Chinese script includes a set of

elaborate ideographic characters for numbers. These are used on financial documents in much the same way the spelled-out numbers are used on checks in English.

- Line breaking. For each Asian language, there is a set of rules for inserting line breaks into text. These have nothing to do with word boundaries.
- Dates and calendars. Japan, Korea and the Moslem world all have alternative calendars that are in common use. Depending on the context in which you are presenting or soliciting a date, you may have to work with an alternative calendar.
- Numbers that identify people. The US has nnn-
nn-
nnn social security numbers. Other countries have a wide variety of identification schemes with varying formats.

Parallel Development

In an ideal world, your development team would already include a few international aces, thus obviating the need for a separate group of experts. But perhaps your company has international developers in a special group or prefers to outsource the work. Acme took that difficulty and exacerbated it even further. To work successfully with parallel development teams, you need good communication and strong source control.

Consultants

Internationalization projects often involve outsourcing. Specialized expertise is needed, often in conjunction with hurry-up schedules. Some consultants will educate your core developers as part of the process. Some won't. Stick with the first kind.

Once your code is internationalized, it has to be localized. That is the process of producing translations of strings and other materials for each target. Unless you work for an enterprise that is so vast that it can afford to have a staff of professional translators, use an outside vendor for localization. Your international developers (in-house or outsourced) have the job of making sure that the necessary materials are easily identified, handed to the localization vendor and reintegrated after translation.

Architecture

Once the requirements are set, the next step is architecture. There are entire books—Ken Lunde's *CJKV Information Processing* (O'Reilly & Associates, 1999) is one—that discuss the various alternatives, so I'll restrict myself to a single example. The most basic question is where to put the strings. In many cases, there are three alternatives: Windows resource files or Java resource bundles, some other sort of message catalog file or a database.

When deciding among these, it's important to question whether you need to access strings from more than one language at a time. If you are implementing a Web server, for example, you may need to grab the strings that apply to a particular client's users, since different clients are in different languages. Some message catalog systems enforce a single, static language selection. One of those systems would be a very poor choice in a Web server. On some operating systems, the run-time library has a single language setting (the locale) for an entire process, and there is no thread-safe way to change it for the life of the process. On such a system, you must either ensure that a particular process handles only requests for a particular language, or you have to substitute a thread-safe mechanism for the system mechanism.

Another important consideration is performance. If you use a database to store strings, beware of introducing a database query latency into a performance-sensitive code path. One good strategy is to store a time stamp in the database that records the last time that the strings were updated. Clients can then maintain a local cache of strings and only retrieve them from the database when they change.

Schedule

Internationalization first shows up in the schedule as the features chosen in the architecture are developed. So far, we have simply added more development tasks to the schedule. Later on, things get more complicated, and many of these things are the actual localization of the code.

Translation and localization take time. Before you can start translating, you have to have a set of strings to translate. You have to freeze the strings. If you can't freeze them altogether, you have to start to track the changes to strings, so that you can send incremental jobs to the translator(s) to keep up with

developments. One way to expedite translation is to start with a glossary. Extract a list of important words and phrases from your code and documentation. Send it off to your best (and perhaps, most expensive) translator. Have it reviewed by an editor. When the time comes to send the bulk of the text off for translation, have the translator work from the glossary.

Quality

Once you send strings off to the translators, it takes them some time to complete the work. In the meantime, you may be wondering how well all your internationalization changes have worked. “Gee,” you may think, “too bad we can’t start testing this stuff yet.” Well, you can. You can pseudo-translate to find and flush out many defects before the translations are available.

In pseudo-translation, you pretend to localize the product. You take the original translatable materials, and you decorate each string with (for example) a few Chinese characters. Then you test the product, looking to make sure that the Chinese characters appear, correctly, in all the right places.

Of course, you must leave room in the schedule for late translation fixes. In spite of the best efforts of translators and editors, mistakes and misinterpretations will turn up in the late stages of testing. Allow time to send them off for corrections.

A Business Imperative

Now that you know the truth about internationalization—it isn’t easy, it isn’t simply a localization project, and you can’t accomplish it on a shoestring—you can focus on the rewards of successfully translating your software for use in other countries. International Data Corporation estimates that worldwide business-to-business e-commerce will grow to \$30 billion by 2001, while by 2002, non-English speakers will make up more than 50 percent of the world’s online population.

With more than half of the world’s Internet users predicted to be non-native English speakers by 2002, going global is not merely a business advantage in the 21st century; it is a business imperative.

Using Unicode to get you there is the most efficient and promising way to ensure your

worldwide engineering process is effective, affordable and rewarding. ■

Details of MBCS Issues

Acme’s code was full of little constructs that were all set to misbehave when presented with text in a Japanese character encoding. Japanese text is usually encoded in an MBCS (“Multi Byte Coding System”). That means that some characters are one byte while some are two. (On some platforms some characters require more than two.) Many common coding clichés don’t work with MBCS text. Code like:

```
for(sp = buff; *sp; sp++) { ... }
```

is not processing characters. It is processing bytes. If it stops in the middle of a character, or compares a single byte to some ordinary ASCII character, it can end up in the middle of a multi-byte character, with disastrous consequences. For example:

```
strchr(buff, '/')
```

can end up returning a pointer into the middle of a character. ■

Dialog Box Woes

The font metrics are different in Asia. If you carefully design a dialog box to fit in 800x600, it won’t fit as soon as you try to run it on an Asian system.

Individual strings may be bigger (overflowing their graphical boundaries) or smaller (leaving ugly white space).

Your layout may embed an English grammatical assumption. The right word order for English is not necessarily the right word order anywhere else. For example:

```
From [EDIT CONTROL] To [EDIT CONTROL]
```

is an ordering of screen text that assumes English grammar; in Korean, the user fields would not fall in that sequence.

In Asian fixed-width fonts, characters come in two sizes: full-width and half-width. The ideographic characters are full-width; the Latin characters are half-width. And then there are special full-width Latin characters. If you have code that sets up neat columns by assuming a matrix of fixed-

width characters, it won't look good until you teach it to count on halves of fingers. ■

Technology

Internationalization technology is a very broad topic. Almost anything that touches text can turn out to have or create internationalization problems. Here is a survey of the major difficulties that many projects encounter.

Third-party components. The most dangerous aspect of any code base, from an international standpoint, is the presence of third-party components. If the component doesn't work right for international text, you have to either get the vendor to fix it, find another vendor, or recreate the functionality for yourself. Any of these can have a long lead-time. Survey your third party components early, and find a strategy for them.

Strings. Most likely, your code has strings seen by human beings. You have to find them. You have to distinguish them from strings that are not seen by human beings. You have to arrange for them to be translated in each release. Most importantly, you have to make sure that ongoing development does not undo all the work of finding the strings and making them available for translation.

MBCS. The classic bug-bear of Asian internationalization is MBCS. These days, most programmers have the idea that writing 8-bit-clean code is a good idea. Characters that occupy more than one byte, and especially variable numbers of bytes, are another story. Almost all code that processes text is unsafe for MBCS text, and requires remediation.

Unicode. Unicode, also known as ISO-10646, is a standard character set for multilingual text. The Unicode Standard defines a very large character set and several character encodings. All commonly-used languages' characters are represented in the character set. Unicode provides a uniform and organized approach to character properties and managing scripts that run right to left.

The character encodings offer you a choice of representations. One choice is UCS-2. In this encoding, all the characters are same size in memory: 16 bits. In UCS-2, you don't have to worry about splitting a character in two. You don't have to worry about the number of elements of the array versus the number of characters. You do have to modify all your code to declare textual data to be represented as

16-bit items instead of 8-bit items. This can be a very expensive conversion to existing code.

To avoid this conversion cost, Unicode defines several alternative character encodings called 'transformation formats.' The important ones are UTF-8 and UTF-8-EBCDIC. Both of these are MBCS formats, and thus subject to MBCS coding problems.

While the transformation formats are MBCS, they are 'friendly' MBCS. Simple ISO-646 (ASCII) characters never occur as a part of multi-byte characters. Thus, many common operations on strings (for example, 'strlen' in C) are valid with UTF-8.

The GUI. Even simple GUIs can encounter serious problems. For an example, consider dialog boxes on Windows. See the sidebar for some of the gory details.

Asian languages have many more characters than fit onto any imaginable keyboard. To allow input, systems provide Input Method Editors. These allow the end-user to construct characters by typing phonetic or other representations of them. If you use standard GUI components, you will find that the IME "just works." If you do your own keystroke handling, you may need some serious effort to co-exist with the IME.

Databases. The major relational databases handle the major international character encodings. However, you may have to make complex or troublesome changes to make use of their support.

Printing. If your product has printer drivers, it is very likely that they will need significant enhancement to print Asian text, even in common printer languages such as PCL and PostScript.