Rosette — Internationalization through message catalogs

Nearly all software presents a language-based user interface: any program containing prompts, responses, menus, status lines or tool-tips falls into this category. Language-based software benefits from translation to the (major) languages of the countries that it ships to. A lot of software that one finds on the shelf today is already available in multiple languages. This white paper presents "Rosette": a library for software developers that brings multi-lingual support into your applications.

Adapting a software product to different languages and cultures is a process called *localization*. To make localization efficient, the first step is to *internationalize* the source codes of a software product. Internationalization involves separating the language and culture-dependent information from the logic. Localization goes beyond translating an application: date, time and currency formats also need to be adapted for an application to be usable in another region.

When speaking about software, the most prevalent products are desktop applications: word processors, accounting programs, database clients, etc. Software is broader than what runs on the desktop, though, and the need for internationalization applies to most other products too. For example, large server applications that deliver their services over the Internet must be prepared to serve in multiple languages at the same time. At the other end of the spectrum, customer goods such as appliances and audio-visual equipment are just too expensive to develop for the national market only. Embedded software, therefore, needs to be internationalized too.

The name Rosette refers to the granite stela found near the small Egyptian town Rashid, which Napoleon's army called "Rosette", and that contains a decree of the priests of Memphis in three scripts: hieroglyphs, Demotic (an Egyptian script used at the time that the stone was carved) and Greek. Known as "pierre de Rosette" or "Rosetta stone", its discovery by the French lieutenant Pierre Bouchard, became symbolic for the break-through, in 1822 by the French Egyptologist Jean-François Champollion, in the deciphering of hieroglyphs.

First question: Why?

Translating an application takes a fair amount of effort, regardless of how you tackle it. It may also require the help of persons that you do not employ in your organization: native speakers of the foreign tongues. Those individuals are probably not familiar with your product when they start the translation, so you also need to assist the translators to get knowledgeable about it. A misunderstanding of the purpose of a menu item can easily lead to an incorrect translation. An example of this was the translation of the menu item "Display options" in the Dutch version of IBM's graphical DOS shell: it was translated to mean "show the options", but what the menu was really about were the "options for the display".[†] A translation error is quite often viewed as a *stupid* error, one that is so obvious that it should have been spotted before deploying the product.

Because of the effort required to translate a product and the risk for wrong translations, some wonder whether translation and localization is worth the effort. Cannot English be seen as the Lingua Franca[‡] of modern times? Or in more popular terms: why can't they just use English? Rather than answering these rhetoric questions, let's just observe that a program that is available in a local language, has a competitive edge over another program which is not. Computer magazines in Europe (and probably elsewhere too) mention the lack of a localized version of an application as a negative note; in product comparisons, points get subtracted.[*]

[†] This is one out of *many* examples where a mistranslation causes confusion, and one out of many where the user is left wondering if the product had been hastily translated and then just "dropped in the market" without further testing.

[‡] *Lingua Franca*, the "language of the Franks", is a medieval mixture of various languages used for trade contacts. This "broken language" allowed people from various regions to communicate at a basic level —much like *broken English* does today.

[*] To quote former German chancellor Willy Brandt: "If I'm selling to you, I speak your language. If I'm buying, dann müssen Sie Deutsch sprechen."

Second question: How?

Once we have decided to make an application available in multiple languages, we have to decide on an approach to tackle this issue. You could, of course, go through the source code of the application, translate all text strings and recompile; you would essentially end up with two applications, one in either language, built from two different sets of source code. However, for sake of maintenance, you will want to choose a method that separates the language from the logic and build all localized versions of the application on a single source code base.

Other decisions are not as clear cut. It might be tempting to pack all texts of all supported languages inside the application as resources. Every separate file that you remove from the package removes a possible "support issue". On the other hand, it also makes it harder to add translations to a product that is already distributed, as the application needs to be rebuild to contain the new translations. And there are other choices to make, like "one file with all translations" versus "a separate file per language" or "indicate each string with a unique numeric key" versus "use strings in one language as the message ID for matching strings in other languages". Which approach fits best depends on the application and your development methods and resources.

It will not come as a surprise that many tools and libraries to assist in translation already exist, and that many of those impose their selection for the various choices on you. Except Rosette: the Rosette library is designed to be flexible and adaptive to different schemes of development. Rosette is a portable, reentrant, multi-platform internationalization library that supports multiple programming languages and has optional Unicode support. Rosette is one of the *very few* internationalization libraries that are suitable for embedded systems, due to its portability and its deterministic behaviour (plus that Rosette does not rely on dynamic memory allocation).

The developer's view

Internationalizing an application with Rosette has two parts: a set of translated messages and a set of functions. The translated messages are the language resource data for the application and this data may be in text or binary form, and either as a separate file or embedded in the application itself. Whatever its format, the message collection is called the "catalog". The functions in the Rosette library manage the catalog and provide the core functionality to retrieve the appropriate message in the selected language. Rosette allows multiple languages in a single catalog and a catalog may be a plain text file: no conversion (or compilation) of the catalog is required before its use.

The catalog file format

The message catalog is a text file, or at least it starts as a text file —it may optionally be compiled to a binary file or resource. In this text file, there is for every message *one* line in each language, with the language identified by a two-letter code, preferably conforming ISO 639 (en = English, fr = French). Some languages have developed several minor variants, such as "British English" and "American English", which are not recognized as separate languages by ISO 639. If so desired, one can use the 2-letter "country code"(ISO 3166) instead of the language code (i.e. us = US-English and gb = The Queen's English). For locale-dependent information, you can use a "tagging" system.

Here is a snippet of a catalog with two messages in three languages:

```
// source: http://www.geocities.com/nodotus/hbglass.html
en: I can eat glass, it does not hurt me
fr: Je peut manger du verre, cela ne me fait pas mal
nl: Ik kan glas eten, het doet me geen kwaad

# source: http://www.trigeminal.com/samples/provincial.html
en: Why can't they just speak English?
fr: Pourquoi, tout simplement, ne parlent-ils pas français ?
nl: Waarom spreken ze niet gewoon Nederlands?
```

In the catalog file, leading white space is on a line ignored; empty lines and lines starting with a double slash ("//") or with a hash sign ("#") are ignored; after the 2-letter language code, the first (optional) space behind the colon is ignored. For every message, one particular language should come before all translations: this is the key language. In this snippet, the key language would be English ("en"). The catalog also supports a few special characters (or escape sequences) for embedded line-breaks or TAB alignment.

To add multi-lingual support to an application, the first step is to go through the source code and mark all hard-coded text (strings) that need to be translated to other languages. Marking such strings is a simple matter of folding the string inside a function; for example, the string `"the quick brown fox"` would become `rsmsg("the quick brown fox")`. Depending on the application, there may be only few or many of such strings. The Rosette manual gives suggestions for performing this stage efficiently.

A Rosette utility, `rsverify`, then creates an initial catalog with only one language. During maintenance, this same utility can also tell you which messages were newly added. This catalog is what you send to the translator, preferably along with the executable application.

The developer has to perform one last step before the application is internationalized: the Rosette library must be added to the project and the source code must also be updated to initialize and open the catalog of choice. Depending on the programming environment, it may be necessary (or convenient) to add a few "wrapper functions" or macros.

Because of the flexibility of Rosette, the process of internationalization with Rosette is variable, too. You may want to add a preprocessing step between the "raw" catalog and the one that your application will ultimately use, and you may want to convert the catalog to a binary file or resource after it comes back from the translators. Depending on how you use Rosette, your application needs to be adapted or extended. The Rosette library provides the routines to do this, and the manual explains in detail how to use the functionality of Rosette to perform a specific task.

After the application has been internationalized and the first translations are available, maintenance has its own chores: messages change or become redundant, and new messages must show up in the catalog. Keeping the "master" catalog up to date becomes a task of its own. To make you catch catalog problems early in the process, Rosette provides effective diagnostics and it can display statistics about the message usage.

The translator's view

The catalog is, in its source form, a text file, in ASCII/ANSI or UTF-8 encoding. The requirements for translating a catalog are *technically* no more than Windows NotePad or any other basic text editor. However, for the sake of consistency and speed of translation, a translator will prefer to process the catalog with specialized *Computer Aided Translation* (CAT) tools. For the purpose of exchanging files between the Rosette library and CAT software, Rosette comes with conversion utilities for comma-separated values (CSV), `gettext` PO and XLIFF formats.

In translating software from a catalog with the isolated messages, one of the pitfalls is that the translator sees the messages out of context. It is clear that depending on the context, a message in one language must sometimes use a different wording. In the introduction of this white paper, we already presented the example "Display options" —a prompt that is ambiguous when seen in isolation. By allowing the ("text file" format) catalog to be reloaded without needing to rebuild the application, a translator may verify the context in which the translated messages occur by simply running the application after copying the newly created catalog to the application's folder.

The language spoken in a country or region is just one part of the "locale" information, albeit an essential part. Other information, such as the formatting of dates, numbers and currencies must be handled according to the local conventions too. These conventions are not tied to the language; for example, numbers are written with a decimal comma in Spain and a decimal period in Mexico —but Spanish is spoken in both countries. Rosette handles locale-dependent information through specially tagged messages. This allows a single catalog to be used for both the Spanish and Mexican locales.

Features and traits

To look up a message in a language, the message must have a kind of unique identifier. Here, you have a choice: Rosette can use the text of the message itself as the "message ID", but it also allows you to use a numeric constant or a symbolic tag. The former solution, text as the ID, is often convenient, but it also means that if the

The intricacies of internationalization

Unfortunately, it is often thought that translating a piece of software involves little more than replacing each "string" in the original application by a string in another language. This is an underestimation. Issues like character sets, word order, rules for singular and plural forms of a message, and formatting dates, wall-clock time, numbers and currencies must all be handled.

Often, parameters need to be inserted in messages —such as names of users or files. Such functionality is already provided in any programming language, but only using a fixed sequence. The Rosette library lets you re-arrange the order of "inserts", as the word order in a sentence may be different across the supported languages. Rosette does this via numbered placeholders in the messages in the catalog file, starting at one. The figure below shows an error message from a catalog with two placeholders and its translation in German.

```
en: Not enough memory to %1 the record %2.
de: Ungenügendem Speicher um Datensatz %2 zu %1.
```

In English, values use a decimal period and thousands are separated by a comma. In the Netherlands and Germany, the decimal separator is a comma and the thousands separator is a period. In France, the decimal separator is also a comma, but the thousands separator is a thin non-breaking space, or none at all, in absence of thin spaces. The grouping of the whole number part of the value does not have to be by 3 digits (thousands separation); in some Asian countries the values are grouped by 4 digits (multiples of ten-thousand). Rosette's number and currency reformatting is quite flexible.

Locale	Example
Great Britain	12,345.67
Netherlands	12.345,67
France	12345,67

Related to formatting messages with numbers is the conditional forming of plural forms. For example, when your application displays "10 matching records" in its status bar for a database query that indeed returns 10 records, you will want it to show "1 matching record" when there is only a single result. Since, in English you obtain the plural form of a word by attaching an "s" to the singular form of the word, programmers have used tricks like the line below, where the placeholder `%s` is replaced by either an "s" or by an empty string:

```
printf( "%d record%s found", count, (count != 1) ? "s" : "");
```

In the above example, the idea is to print the word "record" when the count is 1 and "records" for any other count. Imagine what happens when the target language is "de": the plural form of Datensatz (the German word for "record"), is "Datensätze" —it is certainly *not* "Datensatzs". The umlaut on the second "a" also demonstrates that no general purpose rule will ever construct a plural form from a singular form correctly.

Appending an "s" to a word creates a plural form in French too, but the example is still wrong in a more subtle way: you have to use the *singular* form for a count of zero in French. Also note that the "trick" only works for sentences lacking a finite verb —but these are regarded as grammatically incorrect in France.

The proper solution is to enter the messages with singular and plural forms explicitly in the catalog, and as complete messages, as is illustrated below. The catalog contains the full messages for forms with an explicit count. This need not be limited to zero and one; a few languages, like Arabian and Slovenian, have an additional *dual* form. There is no need to specify the same plural forms for all messages; in this instance, there are exceptions for the counts 0 and 1 in French, but only for 1 in English and German.

```
en   : %1d records are found
en[1]: %1d record is found
de   : %1d Datensätze sind gefunden
de[1]: %1d Datensatz ist gefunden
fr   : %1d lignes sont trouvés
fr[0]: %1d ligne est trouvé
fr[1]: %1d ligne est trouvé
```

The programmer can now use the plural form in the key language ("en" in the above example) and let Rosette choose the appropriate singular or plural form. For example, the earlier `printf` statement becomes:

```
rsfmti(output, sizeof output, "%1d records are found", count, count);
```

message changes, both the application and the catalog must be adapted. With a separate numeric key or tag, you would only have to change the catalog, but at the cost of having to maintain a list of keys/tags that match a message, and both the source code and the catalog must agree on what message goes with each. Numeric keys are somewhat specific to C/C++ (because of the use of the preprocessor) but tags can be used with any language.

In practice, multi-lingual support must often be built into an application which was initially developed without giving internationalization much thought. Obviously, adding multi-lingual support *after the fact* takes more effort than building it in from the beginning —but in many projects this is the proverbial wisdom of hind-sight: very true, but not very helpful. Rosette's feature of using the messages in the original language to look up messages in any of the translations makes the path easier.[†] However, there is also a risk: depending on the context, two messages that would use the same wording in English need to be translated to different messages in the other language. Using the original message is now an ambiguous criterion for looking up translations (assuming that the original language is English).

Rather than completely abandoning the concept of using the original messages for looking up translations,[‡] Rosette allows the conflicting messages to be tagged with a unique symbolic key. The translated message is then looked up through this tag. You may tag *all* messages, but doing so with only the conflicting messages will get you to the finish line earlier.

Rosette is a C library and it comes as a precompiled component (DLL) as well as with full source code. In the Microsoft Windows environment, you can use Rosette with any programming language or development system that can call a DLL. The Rosette manual gives examples for C, C++ Delphi and Basic.

For Microsoft Windows programs, among the first things that you will want to translate are the menus and dialogs. In fact, there exist software translation products that focus solely on the resource data. But even when using a source code oriented library such as Rosette is it straightforward to translate menus and dialogs in an application —plus that it also handles non-resource data, which is off-limits for resource translation programs.

Rosette is a non-intrusive library: it does not require any specific platform or framework and does not force you to switch to a different "string library" or character encoding. When it comes to international character support, there are three options in common use: codepages, Unicode and UTF-8. Rosette can work with all three; it does *not* support character set conversions, however.

Being non-intrusive also brings forth some limitations of the library. By not replacing the string library that your applications already use, Rosette cannot provide support for locale-dependent sort order. Support for directional reformatting (right-to-left scripts, commonly called *bidirectional* languages) is not present either. This is a conscious choice: support for these aspects is independent from message look-up. Rosette can work together with other libraries that provide bidirectional support or sort orders of non-alphabetic scripts.

On the technical front, an important note is that the functions of the Rosette library are reentrant (*except* for the initialization and clean-up functions). The advantage of reentrant functions is that they are implicitly thread-safe. Although you can protect concurrent access to non-reentrant functions with semaphores or critical sections, it also carries a risk of priority inversion (if threads have different priorities). You should be able to do a message lookup without giving it a second thought, and a reentrant library is then the more robust design.

Memory management in Rosette gives you a palette of choices again: you can use static memory, explicitly handled dynamic memory and garbage-collected dynamic memory. The last option is the most convenient for the developer, but it also requires the most resources from the operating system.

The catalog starts as a text file, encoded either in ASCII/ANSI or UTF-8, but you have the option to convert it to a binary file, a binary resource or a C/C++ source code file that allows you to directly compile the catalog into the application. This latter part is convenient for embedded systems that have to do without file system or resource manipulation support. The catalog is "ROM-able", by the way.

The existence of similar products as Rosette was already mentioned. How, then, does Rosette compare with `catgets`, `gettext`, Microsoft's Message Compiler or the wide spread use of INI files with message

[†] Rosette also provides extensive diagnostics and message statistics modes to help developers catch missing translations or changed source messages.

[‡] Rosette supports a mode for using *only* numeric message IDs.

key/translated string entries? The `catgets` library is difficult to use: keeping the numeric message IDs match between the catalog and the source code is a burden. Microsoft's Message Compiler improves it by using symbolic constants, but it is still not as easy as simply using the string in a "key language" as the ID for the message, as do `gettext` and Rosette. Apart from this, Microsoft's design ties the system to C/C++ and, of course, it is only available on Microsoft Windows. The `gettext` library also has a hard link to C/C++ because it redefines the `sprintf()` function family. By not supporting "tags", `gettext` cannot solve conflicts where messages that use the same wording in English need to be split into different messages in other languages. Finally, `gettext`'s message look-up functions are also non-reentrant, because they store information in an internal "cache". Using INI files for translation is indeed simple, but its performance is low and its capabilities are limited: it supports neither message reformatting nor plural forms, both of which are absolutely essential for decent translations.

Résumé

Rosette is a library and a set of tools that assists developers in making their applications multi-lingual. The applications will still need to be translated, by *native speakers* of the other tongues, but with Rosette, that task takes less work and the risk of translation errors is reduced.

As detailed in this white paper, Rosette is general purpose and flexible. It is suitable for embedded systems, as well as software running on large servers. CompuPhase, existing since 1986, has used many of the common approaches to internationalize software products before arriving at Rosette. The Rosette library builds on a vast "hands-on" experience in building *and maintaining* multi-lingual applications —either from the start or adding multi-lingual support to an application "after the fact". Rosette's goal is to make message translation convenient for both the programmer and the translator, while supporting several platforms and multiple programming languages.

Rosette

- For all 32-bit versions of Microsoft Windows (including Windows 98).
- Unicode, UTF-8 and ANSI/ASCII are supported.
- Interface files for C/C++, Delphi, Visual Basic and others.

INFORMATIE-TECHNOLOGISCH BUREAU
Compu Phase
AUTOMATISERING

1e Industriestraat 19-21 — 1401 VL — Bussum —The Netherlands

Telephone:	+31 (0)35 6939261
Facsimile:	+31 (0)35 6939293
Internet:	http://www.compuphase.com
	info@compuphase.com