# Where ARE Those Strings???

Externalizing UI Strings from Code—
Tips for the Windows Developer

By

**John White**
www.ventajamarketing.com

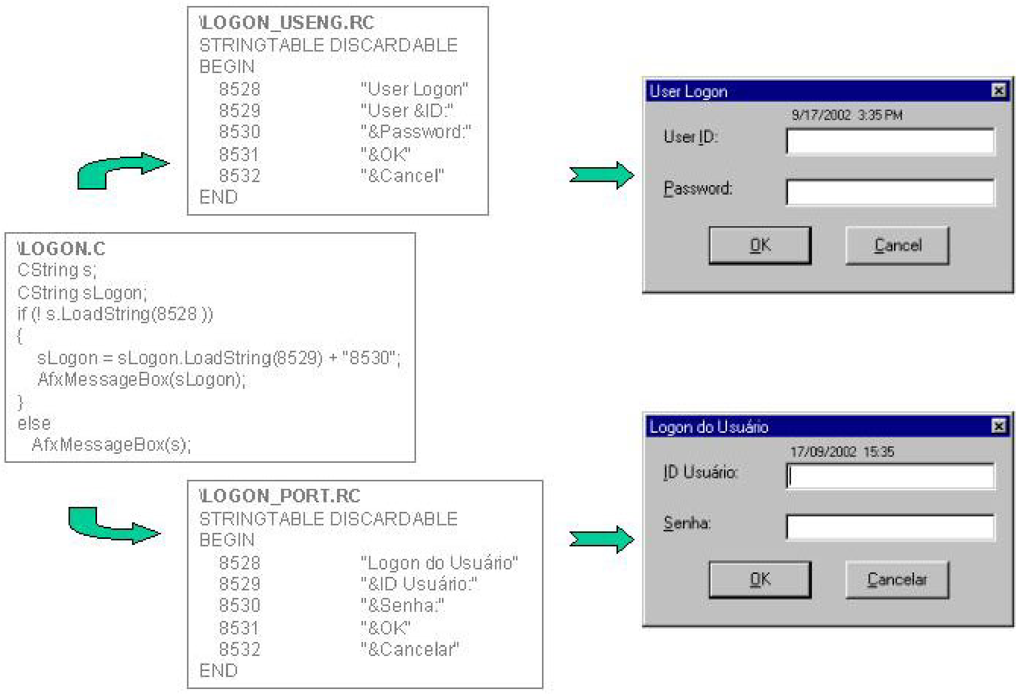
- Why are there untranslated strings in my localized product?
- Why didn't the translators find them?
- Where do untranslated strings like to hide in Visual Studio apps?

Sooner or later, most of us who work with international software products find untranslated strings popping up in the middle of our translated user interface (UI). These don't usually prevent the product from running properly, but they are most unwelcome, and we devote a lot of time to eliminating them so that our customers don't find them and complain about them.

As the Pareto Principle suggests, we have taken care of 80% of the strings in the product with only 20% of our effort, but the last 20% of the strings seems to require 80% of our effort. In this article, we'll explore the process of eradicating untranslated strings.

## Background

To properly *localize* a software product for the needs of particular region, you'll find it best in the long run to first *internationalize* the software, as in Figure 1, so that no matter how you need to change it for user interface or business logic, the code base at the core of the product is always the same (sometimes called a *single worldwide binary*). You then *externalize* those features and characteristics that can change from one region (or *locale*) to another—language, color scheme and accounting standards, for example—to separate *resource files* that vary from locale to locale, then call them from your single, common code base.



```
\LOGON_USENG.RC
STRINGTABLE DISCARDABLE
BEGIN
   8528          "User Logon"
   8529          "User &ID:"
   8530          "&Password:"
   8531          "&OK"
   8532          "&Cancel"
END
```

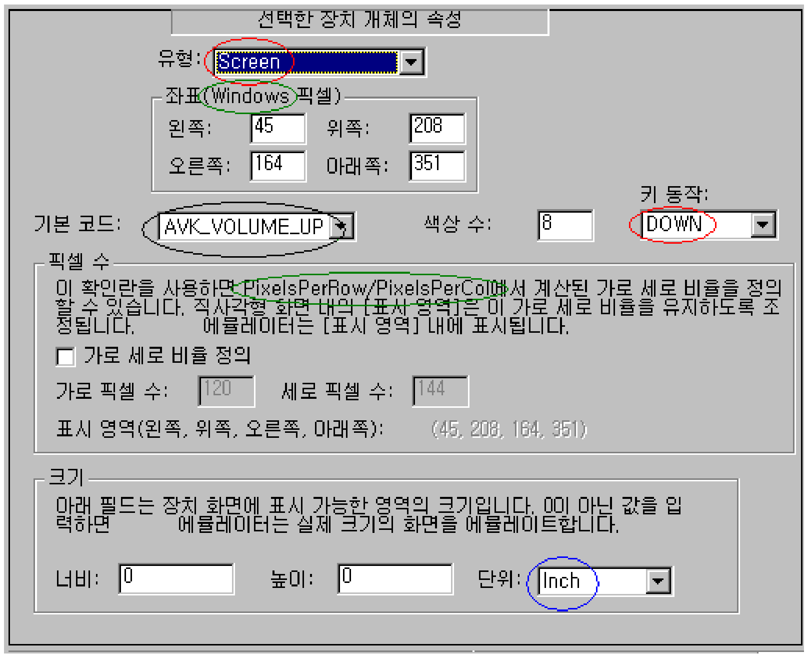
```
\LOGON.C
CString s;
CString sLogon;
if (! s.LoadString(8528 ))
{
   sLogon = sLogon.LoadString(8529) + "8530";
   AfxMessageBox(sLogon);
}
else
  AfxMessageBox(s);
```

```
\LOGON_PORT.RC
STRINGTABLE DISCARDABLE
BEGIN
   8528          "Logon do Usuário"
   8529          "&ID Usuário:"
   8530          "&Senha:"
   8531          "&OK"
   8532          "&Cancelar"
END
```

**Figure 1**

While it has become easier and easier for developers to internationalize their products in this way, rarely does anybody do it completely right the first time. And, while there are techniques for ensuring that the product has been properly internationalized, most companies new to localization will probably not devote resources to them. So, what we've seen time and again is this process:

- the resource files go out for translation and come back in
- we rebuild the product and perform QA
- we find, in the middle of an otherwise translated screen, untranslated strings
- we take a screenshot, as in Figure 2, and submit it as a bug


선택한 장치 개체의 속성

유형: Screen

좌표(Windows 픽셀)

왼쪽: 45 위쪽: 208

오른쪽: 164 아래쪽: 351

키 동작: DOWN

기본 코드: AVK_VOLUME_UP 색상 수: 8

픽셀 수

이 확인란을 사용하면 PixelsPerRow/PixelsPerCol에서 계산된 가로 세로 비율을 정의할 수 있습니다. 직사각형 화면 내의 [표시 영역]은 이 가로 세로 비율을 유지하도록 조정됩니다. 에뮬레이터는 [표시 영역] 내에 표시됩니다.

☐ 가로 세로 비율 정의

가로 픽셀 수: 120 세로 픽셀 수: 144

표시 영역(왼쪽, 위쪽, 오른쪽, 아래쪽): (45, 208, 164, 351)

크기

아래 필드는 장치 화면에 표시 가능한 영역의 크기입니다. 0이 아닌 값을 입력하면 에뮬레이터는 실제 크기의 화면을 에뮬레이트합니다.

너비: 0 높이: 0 단위: Inch

**Figure 2**

## I. Did the translator miss them (or choose to skip them)?

This is the first question to ask, because you can usually answer it by examining the translated resource file and discussing your findings with the translator. So, you open the file in a simple text editor and search for each English string, thereby finding

```
GROUPBOX        "좌표(Windows 픽셀)",IDC_STATIC,59,83,120,45
```

and `LTEXT           "이 확인란을 사용하면 PixelsPerRow/PixelsPerCol에서 계산된` (the strings circled in green above). You contact the translator and learn that:

- "Windows" is still in English because the translator assumed that it refers to the operating system;
- "PixelsPerRow/PixelsPerCol" is still in English because most localization translators know that multiple words without a break are usually a single technical term.

So, while the translator did not miss any of these strings, s/he correctly chose to leave them in English. (Your translators should already know this, by the way, either because you told them, or because they're familiar with your product.)

However, you cannot find the other untranslated strings in the resource file, so you move to the next question.

## II. Are they meant to remain that way?

Some strings are meant not to be translated, and so have deliberately not been externalized to resource files. The string may be a *literal*, which the code expects to find exactly as is. In the example "\winnt\system\", if "system" were to be translated, the software would probably not recognize the path, and malfunction. Or, if your users are software developers or system administrators, for instance, they may be accustomed to certain English terms, such that there is no point in translating them.

You determine from the developers that, even though the string circled in black appears in the UI, "AVK_VOLUME_UP" (and all others in that list) should be left untranslated. They expect that users will tolerate and understand these somewhat English strings, and besides, these are the names which the code would look for during execution, so translating them would cause the software to malfunction.

The strings circled in red and blue, however, should appear in Korean, so you continue the hunt.

## III. Are they still in our code base?

The next place to look is in the code base, in case the strings eluded your developers' first attempts to find them. Since most strings are enclosed in quotation marks, your developers will probably search on `"` to find any remaining strings, although this will yield lots of false positives. In a Unicode-enabled product, developers will target `_T("` and `_T` and `L"` for more precise searches.

While this is the right way to look in the right place, in this case, your search does not turn up the strings you seek.

## IV. Do they come from NIH software (plug-in's, included files, etc.)?

Although it might not occur to them until you mention it, your developers may realize that some strings come from *NIH* (not invented here) parts of the product.

For many products, the most conspicuous piece of NIH is the installer. Few companies go to the trouble of creating their own installers any more because well-behaved, off-the-shelf installers are already available to developers, and their strings are properly externalized and accessible. Not all NIH is that clean, however, and there may be plug-ins or include-files or controls—calendar controls have been the worst offenders, in our experience—with hard-coded strings you cannot reach.

It may take you several conversations to find the developer who remembers, “Oh, yes, that control in the dialog box is from a toolkit somebody bought six or seven years ago. We don’t have the source code for it, and nobody is supporting it anymore. Other than that, it works fine in English.”

You determine, then, that “Inch” (circled in blue) belongs to an NIH control used to convert between English and metric units, and you begin the long process of contacting its author for an internationalized version. If you are unsuccessful, you can shop for another, similar control; try to convince Engineering to write its own, new control (properly internationalized, of course); or leave it in English for now and resolve the matter in the next version.

## V. Do they come from the operating system?

Imagine that you are reviewing the first draft of your Chinese product, and that you see the screen in Figure 3:

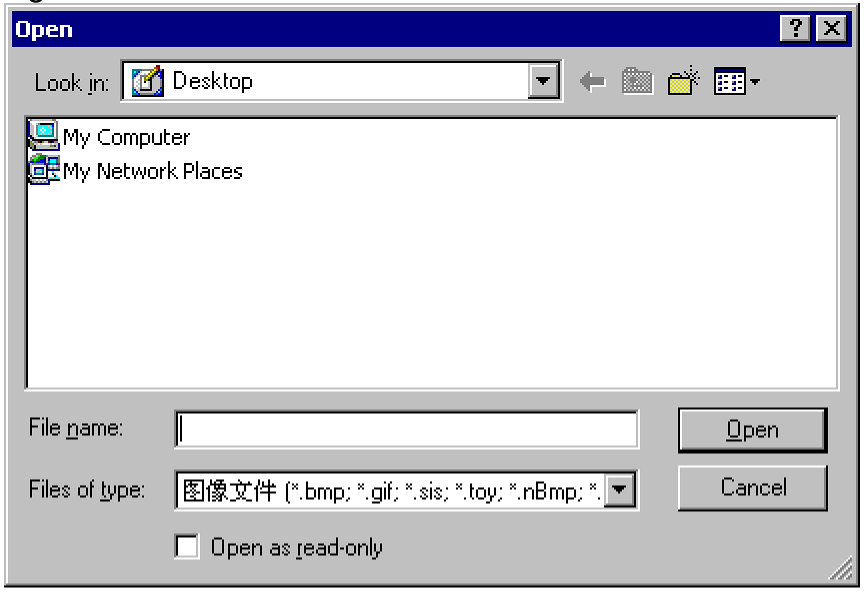


**Figure 3**

You may be tempted to conclude that “Look in:”, “File name:”, “Files of type:”, etc. are untranslated strings, and begin your inquiry all the way back at Question I. . Another, simpler way of asking this question is, “Does the problem go away when we run the Chinese software on a Chinese operating system?”

Some common dialogs and screens come with the operating system and are not really part of your product. While your product makes use of them, your developers do not need to create them, but merely call them at certain places in the code base. Figure 3 is an example of this. Most of the strings here appear in English because you're running the software on an English operating system (with just enough support to properly display your Chinese characters). You'll be better served by working on a Chinese testbench, in which case this dialog box will resemble Figure 4, where the only remaining English text is in the filename extensions (.bci, .bmp, etc.), which, of course, should not be translated anyway.

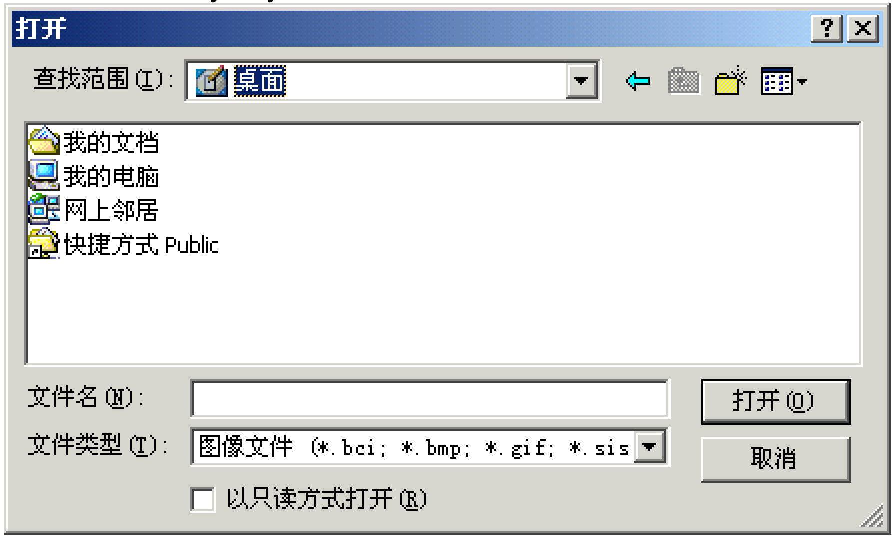


**Figure 4**

To return to Figure 2, you determine that, even on a Korean operating system, the strings circled in red—"Screen" and "Down" and the other items in those drop lists—are still in English and need to be translated.

## VI. Where ARE those strings??

Finally, you convince a developer to help you by loading the Korean resource file and project into the resource editor in which it was originally written (an extraordinary step, but you'll try anything at this point). The developer is able to see quite clearly that "Screen" and "DOWN" are still untranslated, and he insists that they must be in the resource file. He opens the resource file and realizes that he, too, is unable to find the strings. He then looks through the resource file for the name of that particular drop list—IDC_COMBO_OBJECTTYPE—and finds:

```
IDD_QSKINATTR DLGINIT
BEGIN
    IDC_COMBO_OBJECTTYPE, 0x403, 7, 0
0x6353, 0x6572, 0x6e65, "\000"
```

It's actually a rather frightening thought, but the untranslated strings are preserved in hexidecimal values in this DLGINIT: `0x6353` resolves to "cS", `0x6572` resolves to "er", and `0x6e65` resolves to “ne”, which unscrambles to “Screen”. These strings were indeed in the resource file all along, but the translator’s tools didn’t find them, so the translator never saw them.

Your short-term solution is to use the resource editor itself to place the Korean translations of “Screen” and “DOWN” into the resource file. Your long-term solution is to encourage your developers to stay away from DLGINIT types, and to avoid prepopulating any controls with string literals. Rather, they should programatically populate combo boxes and droplists using strings defined in the resource files.

## VII. Summary

By now you’ve exhausted the most likely places in which untranslated strings can hide, and successfully answered your questions. Be prepared to follow this process again, as your company builds and acquires new software for you to localize.

## VIII. Next Steps

You’ve learned something from this paper, haven’t you? You’d like a strong process, plan and player for your localization project, wouldn’t you?

To give your localization effort every chance of succeeding:

1. Help yourself to other resources on our Web site.
2. Become as conversant in localization terminology as you can.
3. Contact us for a **free assessment** of your project, before you paint yourself into any corners.

John White of venTAJA Marketing (johnw@ventajaNOSPAMmarketing.com) offers localization project management and international product management for technology companies, and has managed internationalization and localization projects since 1992. He sometimes wakes up in the middle of the night remembering another place in which untranslated strings can hide. Apul Nahata of Qualcomm Incorporated also contributed to this article.