

Localizing your .NET Application

Venkat Subramaniam

venkats@agiledeveloper.com

<http://www.agiledeveloper.com/download.aspx>

Abstract

Localization or Internationalization (I18N as it is sometimes called) is the process of blending an application to the local culture of the end user. An application needs to accommodate differences in language, currency, calendar, time and culture sensitive color and messages. Further this needs to be accomplished without having to change the code. In this article we address the capabilities of and facilities provided in .NET for localization.

From across the continent

As I type this article, I am over 37000 feet (should I say over 11,000 meters) above Europe, traveling on teaching assignments in Belgium, the United Kingdom and The Netherlands. While the visit was to teach OO Paradigm and .NET to some wonderful people here, it gave me a bit of exposure to the life in Europe as well. The language was a bit of a challenge in the restaurants in Belgium. It took me several minutes to figure out how to type the “@” on the keyboard I had to use at the Hotel to access email! The Brits speak different English and drive on the wrong side of the streets as well ☺ (wink at my British friends). (Sorry, no offense, but it drove me nuts to be greeted with “Are you alright now?” Of course, for their share the Brits pointed out a number of ways the Americans irked them as well) Having grown up in India, it was a week of some re-living the spellings and phrases I had almost forgotten, having spent significant time of my adult life in the US: Colour, programme, to let, lift and taxi. Of all the times I have offered the .NET course, I found those folks to be the most interested in localization as well. I heard comments like “we need to display in French, Italian, Dutch, ...”

Localization without code change

I am sure as a programmer you would readily agree that developing a different code base for each language is not a viable option. The application has to function as the same, for most part, while presenting information to the user based on their specific locale. It should be noted that some applications have to function differently based on some locale. For instance, an application that helps a citizen file his/her taxes would obviously have to take care of the specific law in the region and it may not even make sense to use large portion of the same code base (or does it?) We are here interested in application that has to function consistently in different culture, like may be simulator for the process industry, an internet browser and a scientific calculator. These applications, however, need to display the information and take the user’s input based on the differences in these locale and culture. How effectively can we realize this without being forced to maintain different sets of code?

Facilities in .NET

The .NET Framework and Visual Studio has a number of facilities to help us localize our application: CultureInfo class, Resource files and ResourceManager class, tools to

manage resource files, satellite assembly, Visual Studio support and related tools. In this article we will discuss each of these.

CultureInfo class

A CultureInfo class provides information about specific cultures. It gives you details including name, country/region, calendar, objects to help you with formatting date, etc. The name is in the form of <language>-<country/region>, where the language is a lowercase code and the country/region is a upper case code, each two letters in length (although a few are three letters like the code for Konkani (India) is kok-IN and the code for Divehi (Maldives) is div-MV). For example, en-US stands of English and United States, while es-MX stands for Spanish and Mexico. Similarly fr-FR stands for French and France while fr-CA stands for French and Canada. The CultureInfo class is part of the System.Globalization namespace. The following code snippet displays all supported locales:

```
string format = "{0, -10} {1, -10} {2, -10}";

Console.WriteLine(format, "Name", "Parent", "Display Name");
foreach(CultureInfo culture in
    CultureInfo.GetCultures(CultureTypes.AllCultures))
{
    Console.WriteLine(format, culture.Name,
        culture.Parent, culture.DisplayName);
}
```

Part of the output produced by the above code is shown below:

Name	Parent	Display Name
ar		Arabic
ar-SA	ar	Arabic (Saudi Arabia)
ar-IQ	ar	Arabic (Iraq)
ar-EG	ar	Arabic (Egypt)
ar-LY	ar	Arabic (Libya)
ar-DZ	ar	Arabic (Algeria)
ar-MA	ar	Arabic (Morocco)
ar-TN	ar	Arabic (Tunisia)
ar-OM	ar	Arabic (Oman)
ar-YE	ar	Arabic (Yemen)
ar-SY	ar	Arabic (Syria)
ar-JO	ar	Arabic (Jordan)
ar-LB	ar	Arabic (Lebanon)
ar-KW	ar	Arabic (Kuwait)
ar-AE	ar	Arabic (U.A.E.)
ar-BH	ar	Arabic (Bahrain)
ar-QA	ar	Arabic (Qatar)
bg		Bulgarian
bg-BG	bg	Bulgarian (Bulgaria)
ca		Catalan
ca-ES	ca	Catalan (Catalan)
zh-CHS		Chinese (Simplified)
zh-TW	zh-CHT	Chinese (Taiwan)
zh-CN	zh-CHS	Chinese (People's Republic of China)
zh-HK	zh-CHT	Chinese (Hong Kong S.A.R.)
zh-SG	zh-CHS	Chinese (Singapore)
zh-MO	zh-CHS	Chinese (Macau S.A.R.)
zh-CHT		Chinese (Traditional)

The above code output more than 200 cultures. The cultures are grouped into invariant culture (culture insensitive – like “”), neutral culture (associated with a language but not a country/region – like “fr”) and specific culture (associated with a language and a country/region – like fr-FR). Observe in the output shown above, however, that zh-CHT (traditional Chinese) and zh-CHS (simplified Chinese) are neutral cultures. The parent of a specific culture is a neutral culture and the parent of a neutral culture is the invariant culture.

Testing your code for different culture:

The .NET CLR determines the appropriate culture to use for your application. However, relying on just this makes it harder to test an application for different culture. One possible way to address this is to specify what culture to use in the configuration file and instruct your application to use that culture. Note, this is purely to help with testing and I am not suggesting that your production code will look up the configuration file to determine culture. The following is a sample configuration file and the code that reads it and uses that culture.

Configuration file (saved in App.config file in VS.NET 2003 – studio creates applicationname.exe.config upon project build):

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <appSettings>
    <add key="CultureToUse" value="es-MX" />
  </appSettings>
</configuration>
```

Code snippet:

```
Console.WriteLine("Culture at start: {0}",
    Thread.CurrentThread.CurrentUICulture.DisplayName);
string specifiedCulture =
    ConfigurationSettings.AppSettings["CultureToUse"];

if (specifiedCulture != null)
{
    Thread.CurrentThread.CurrentCulture =
        new CultureInfo(specifiedCulture);
    Thread.CurrentThread.CurrentUICulture =
        new CultureInfo(specifiedCulture);
}

Console.WriteLine("Culture now: {0}",
    Thread.CurrentThread.CurrentUICulture.DisplayName);
```

The output from the program is shown below:

```
Culture at start: English (United States)
Culture now: Spanish (Mexico)
```

.NET classes are culture aware. For instance, let us modify the above code to print the Date. Here is the modified code:

```
Console.WriteLine("Culture at start: {0}",
    Thread.CurrentThread.CurrentUICulture.DisplayName);
Console.WriteLine(DateTime.Now.ToLongDateString() + "-" +
    DateTime.Now.ToLongTimeString());
string specifiedCulture =
ConfigurationSettings.AppSettings["CultureToUse"];

if (specifiedCulture != null)
{
    Thread.CurrentThread.CurrentCulture =
        new CultureInfo(specifiedCulture);
    Thread.CurrentThread.CurrentUICulture =
        new CultureInfo(specifiedCulture);
}

Console.WriteLine("Culture now: {0}",
    Thread.CurrentThread.CurrentUICulture.DisplayName);
Console.WriteLine(DateTime.Now.ToLongDateString() + "-" +
    DateTime.Now.ToLongTimeString());
```

And the output is:

```
Culture at start: English (United States)
Saturday, March 20, 2004-7:08:31 AM
Culture now: Spanish (Mexico)
S abado, 20 de Marzo de 2004-07:08:31 a.m.
```

Main assembly and Satellite assembly

When you build a .NET application, you essentially are creating an assembly (either a dll or an executable). An assembly is said to be a “Main Assembly” if it contains non-localized executable code and the resource files for a single neutral or default culture (this is the fallback culture as discussed in the next section). An assembly is said to be a satellite assembly if it contains resources for a single culture, but does not contain any executable code. When developing an application for localization, place the default fallback culture resources in the main assembly. For each culture you would like to support, create a satellite assembly and place the respective resources in it. The resource files you place in the assembly must follow a naming convention: ResourceFileName.*language-country/region*.resx. For example, strings.en.resx, strings.en-US.resx, strings.fr.resx, strings.fr-FR.resx.

Resource files, ResourceManager and Resource Fallback

Resource files:

The resource file contains the culture specific information. For each culture supported, create one resource file. We will look at a simple example that supports a few cultures. First let us create a resource file named strings.resx. Here are the steps to create it. Right click on the project in solutions explorer within Visual Studio (2003). Click on Add | Add

New Item.... Select “Assembly Resource File” and enter the name as strings.resx. Modify the resource file as follows:

Data for data					
	name	value	comment	type	mimetype
▶	welcome	????	(null)	(null)	(null)
	bye	????	(null)	(null)	(null)
	thanks	????	(null)	(null)	(null)
*					

Now we will create three more resource files as shown below:
strings.en.resx

Data for data					
	name	value	comment	type	mimetype
	welcome	Hello	(null)	(null)	(null)
▶	thanks	Thank you!	(null)	(null)	(null)
*					

strings.en-US.resx

Data for data					
	name	value	comment	type	mimetype
▶	welcome	Hi	(null)	(null)	(null)
	bye	Have a nice day!	(null)	(null)	(null)
*					

strings.fr.resx

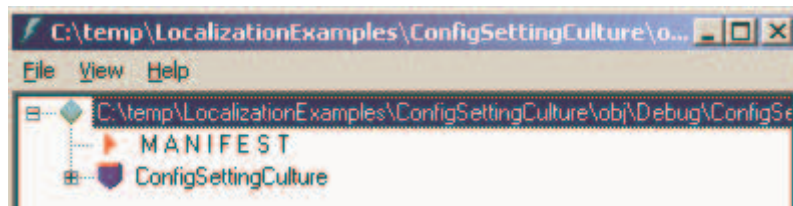
Data for data					
	name	value	comment	type	mimetype
	welcome	Bonjour	(null)	(null)	(null)
▶	thanks	Merci	(null)	(null)	(null)
*					

strings.resx is the default resource file (where I have placed ??? as the values, it is better to put some text in there that would be meaning full). strings.en.resx is a resource file for the neutral culture for English, and strings.en-US.resx is for the English – United States. You may open the actual file and view its contents – these resource files store information as XML. These XML resx files are converted into resource files and placed into the appropriate assembly (main or satellite).

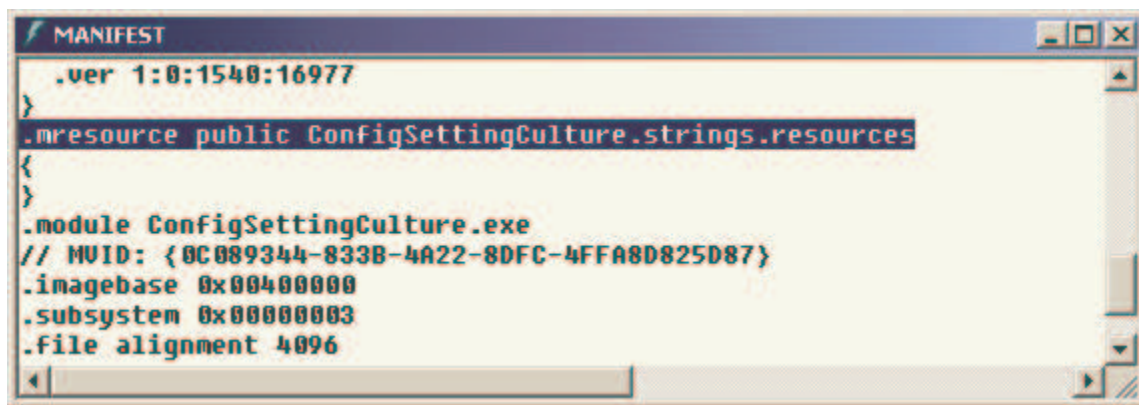
Without writing any further code, let us compile this project. Now look in the bin\Debug (or bin\release if building in release mode):

Name	Size	Type
en		File Folder
en-US		File Folder
fr		File Folder
ConfigSettingCulture.exe	16 KB	Application
ConfigSettingCulture.exe.config	1 KB	Web Configuration file
ConfigSettingCulture.pdb	14 KB	Program Debug Dat...

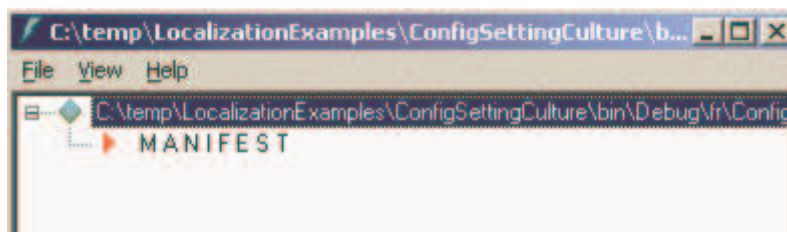
Note that three directories have been created, one per culture we specified in the resource files. A look at what's in the directories en, en-US and fr will reveal the presence of a file named ConfigSettingCulture.resources.dll in each one of them. These files are the satellite assemblies while the ConfigSettingsCulture.exe in the debug directory (shown above) is the main assembly. Let's examine the main assembly using ILDASM (IL Disassembler):



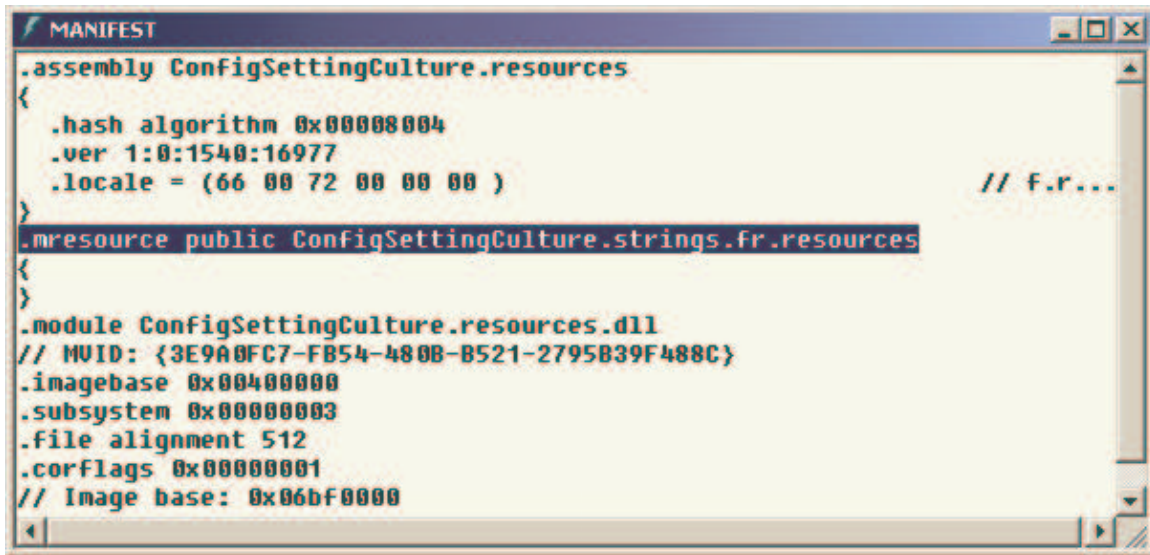
Double clicking on MANIFEST shows us:



Let's take a look at one of the satellite assemblies, say the one in the fr directory:



As can be seen, it does not contain any executable code. Double clicking on the MANIFEST shows the following:



```
MANIFEST
.assembly ConfigSettingCulture.resources
{
  .hash algorithm 0x00008004
  .ver 1:0:1540:16977
  .locale = (66 00 72 00 00 00 ) // f.r...
}
.mresource public ConfigSettingCulture.strings.fr.resources
{
}
.module ConfigSettingCulture.resources.dll
// GUID: {3E9A0FC7-FB54-480B-B521-2795B39F488C}
.imagebase 0x00400000
.subsystem 0x00000003
.file alignment 512
.corflags 0x00000001
// Image base: 0x06bf0000
```

ResourceManager class:

The ResourceManager class provides easy access to the resources that are specific to a culture. The GetString method of the ResourceManager either gets you the mapping value for the given key for the culture of the current thread or for the CultureInfo given as an argument. The ResourceManager belongs to the System.Resources namespace. Now, take a look at the code below:

```
public static void setCulture(string specifiedCulture)
{
    Thread.CurrentThread.CurrentCulture =
        new CultureInfo(specifiedCulture);
    Thread.CurrentThread.CurrentUICulture =
        new CultureInfo(specifiedCulture);
}

public static void display(string specifiedCulture)
{
    Console.WriteLine("---->" + specifiedCulture);
    setCulture(specifiedCulture);
    ResourceManager resourceMgr = new
        ResourceManager("ConfigSettingCulture.strings",
            //ConfigSettingsCulture is the default namespace
            //of the project in which this code resides
            System.Reflection.Assembly.GetExecutingAssembly());
    Console.WriteLine(resourceMgr.GetString("welcome"));
    Console.WriteLine(resourceMgr.GetString("thanks"));
    Console.WriteLine(resourceMgr.GetString("bye"));
}

static void Main(string[] args)
{
    display("en-US");
    display("en-GB");
    display("fr-FR");
    display("es-MX");
}
```

The output from the above code is shown below:

```
---->en-US
Hi
Thank you!
Have a nice day!
---->en-GB
Hello
Thank you!
????
---->fr-FR
Bonjour
Merci
????
---->es-MX
????
????
????
```

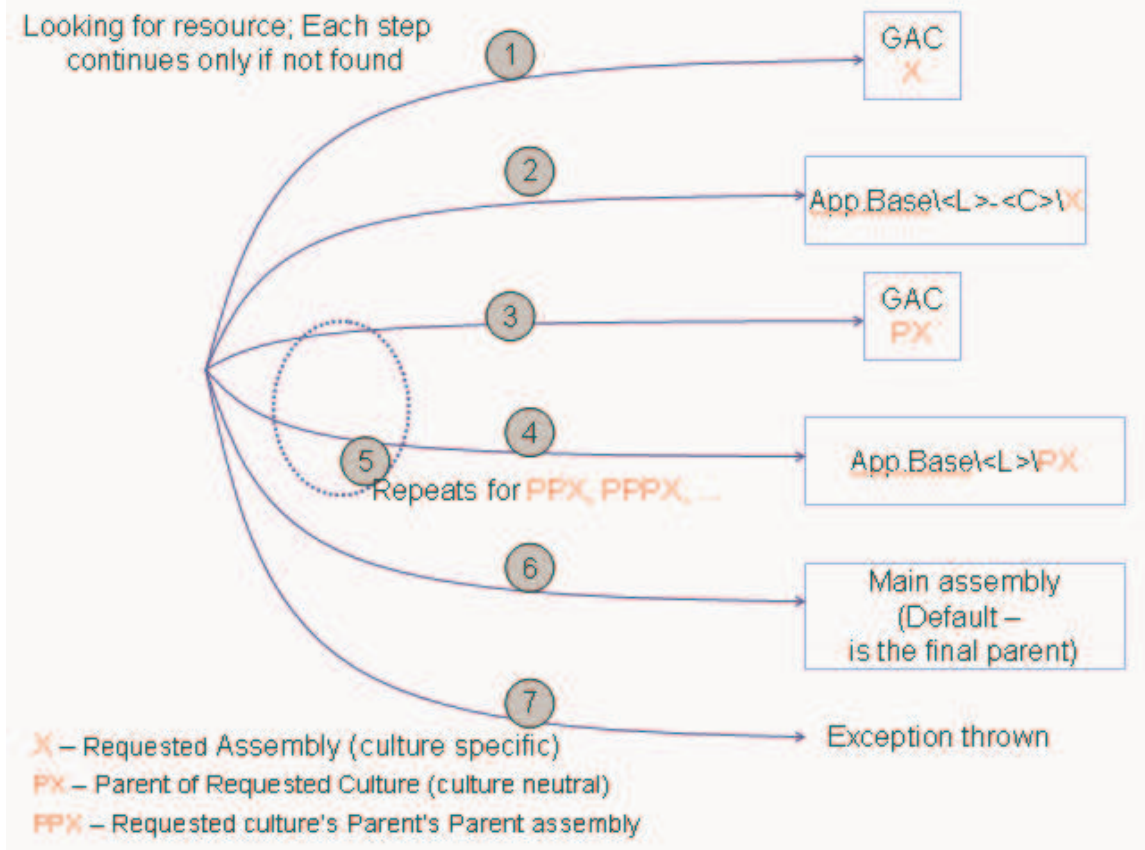
From the output we can see that for en-US, it took what's specified in the strings.en-US.resx file for welcome and bye (culture specific file). However, for "thanks," it took the content from the strings.en.resx file (culture neutral file). For en-GB (United Kingdom), it took what's specified in the strings.en.resx file (culture neutral file) for welcome and thanks. However, since that file does not contain a value for "bye," it took the value from the strings.resx (default – culture invariant) file. In the case of fr-FR, since a culture specific resource is not present, it uses the culture neutral resource for it as well. However, in the case of en-MX, since neither culture specific nor culture neutral file is present, it uses the culture invariant file. If a string is not present even in the culture invariant file, then the ResourceManager returns a null string. For other types of resources, an exception may be thrown.

Resource Fallback:

When a resource is requested, the resource associated with that specific culture is sought. If that resource is not found, then the resource associated with the neutral culture is sought and used if found. If that is not found, then the resource embedded in the main assembly is used. This so called resource fallback is done by the ResourceManager. This is illustrated in the figure below.

Why three levels:

The culture neutral assembly will provide what is common to different variations of a language. For instance, the fr assembly would contain the words from French. Any variations that people in France may have will go into the fr-FR resource file. Similarly for a French speaking Canadian, the overrides or variations will go into the fr-CA file. But, why have the default – invariant file though. Is it better to display some thing than throwing an exception? If you think so, you can put the resource information in the default file then. Instead of the application failing, it would display the default information.



Resource Fallback

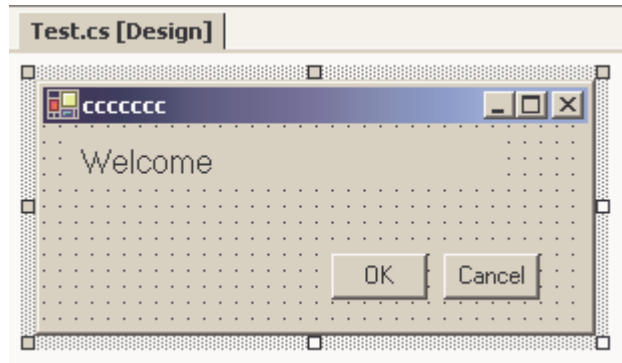
Tools to manage resource files

When we created the different resx files in the above example, these files were first converted into resources files and then compiled into a satellite assembly. We used Visual Studio to compile the project. However, much like csc or vbc are used to compile a C# or VB.NET code, two tools are used in the above process: Resgen.exe and AL.exe. The Resgen tool can either take a pure name = value pair specified text file or an XML file (.resx file) and converts it into a resources file. The Assembly linker (AL.exe) is used to compile the resources into the appropriate satellite assembly.

Visual Studio support and related tools

While the above example was pretty easy to write, how does one handle the locale specific information in a Windows Application? We may take the approach similar to above and write the code in a Form. When we are about to display a text on a Form, we can create an object of ResourceManager and call GetString on it. However, there are other issues to be resolved as well. What if the length of the string in one language is larger than the length of the string in another language? We may have to adjust the size of the label or button accordingly. It was brought to my attention by some folks in UK that the Welsh language is one of those with very long words (I am told that new words were formed by concatenating existing words – so if your dialog has to display the name of your town and what if that town happens to be “Llanfairpwllgwyngyllgogerychwyrndrobwyll-llantysiliogogoch”!). So, how do we handle these situations?

Visual Studio comes with some very nice feature to help us with that. Let's take a look at it with an example. We will first write a simple application that displays in English. The WinForm is shown below:



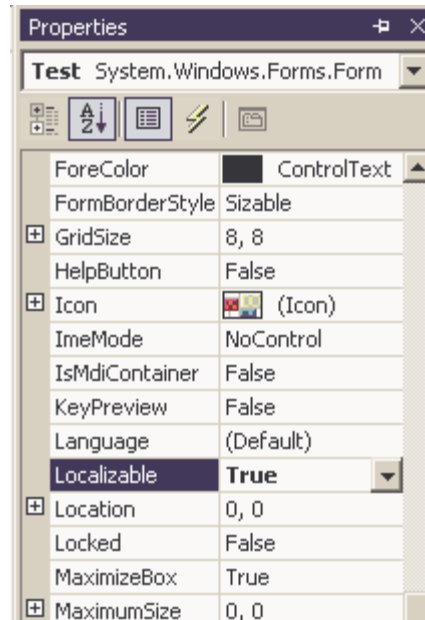
We have not written any code in the Form. Now, this application is not localized yet. Let's take a look at the InitializeComponent function in the Test.cs file:

```
private void InitializeComponent()
{
    this.welcomeLabel = new System.Windows.Forms.Label();
    this.CancelButton = new System.Windows.Forms.Button();
    this.OKButton = new System.Windows.Forms.Button();
    this.SuspendLayout();
    //
    // welcomeLabel
    //
    this.welcomeLabel.Font = new System.Drawing.Font("Microsoft Sans
Serif", 12F, System.Drawing.FontStyle.Regular,
System.Drawing.GraphicsUnit.Point, ((System.Byte)(0)));
    this.welcomeLabel.Location = new System.Drawing.Point(16, 8);
    this.welcomeLabel.Name = "welcomeLabel";
    this.welcomeLabel.Size = new System.Drawing.Size(216, 23);
    this.welcomeLabel.TabIndex = 0;
    this.welcomeLabel.Text = "Welcome";

```

...

As mentioned earlier, we may use the ResourceManager and implement the code to display different language specific text. Let's see what alternative Visual Studio offers. Select the Form and right click on it and click on Properties. In the Properties window, select the "Localizable" property and change it to true as shown below:



What did this do? Examining the Test.cs file we see that quite a bit of change has been made in the InitializeComponent function:

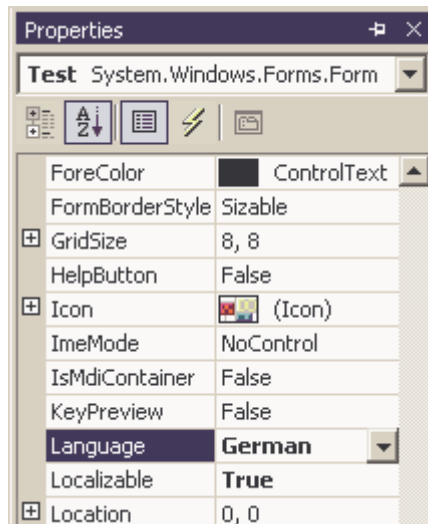
```
private void InitializeComponent()
{
    System.Resources.ResourceManager resources = new
System.Resources.ResourceManager(typeof(Test));
    this.welcomeLabel = new System.Windows.Forms.Label();
    this.CancelButton = new System.Windows.Forms.Button();
    this.OKButton = new System.Windows.Forms.Button();
    this.SuspendLayout();
    //
    // welcomeLabel
    //
    this.welcomeLabel.AccessibleDescription =
resources.GetString("welcomeLabel.AccessibleDescription");
    this.welcomeLabel.AccessibleName =
resources.GetString("welcomeLabel.AccessibleName");
    this.welcomeLabel.Anchor =
((System.Windows.Forms.AnchorStyles)(resources.GetObject("welcomeLabel.
Anchor")));
    this.welcomeLabel.AutoSize =
((bool)(resources.GetObject("welcomeLabel.AutoSize")));
    this.welcomeLabel.Dock =
((System.Windows.Forms.DockStyle)(resources.GetObject("welcomeLabel.Doc
k")));
    this.welcomeLabel.Enabled =
((bool)(resources.GetObject("welcomeLabel.Enabled")));
    this.welcomeLabel.Font =
((System.Drawing.Font)(resources.GetObject("welcomeLabel.Font")));
    this.welcomeLabel.Image =
((System.Drawing.Image)(resources.GetObject("welcomeLabel.Image")));
```

```

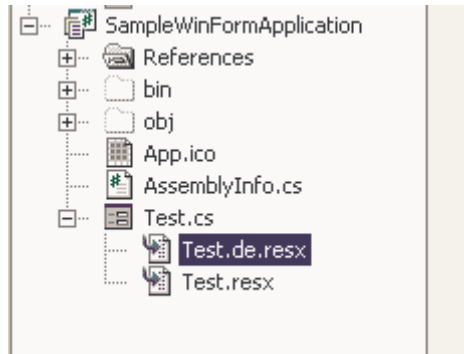
        this.welcomeLabel.ImageAlign =
((System.Drawing.ContentAlignment)(resources.GetObject("welcomeLabel.Im
ageAlign")));
        this.welcomeLabel.ImageIndex =
((int)(resources.GetObject("welcomeLabel.ImageIndex")));
        this.welcomeLabel.ImeMode =
((System.Windows.Forms.ImeMode)(resources.GetObject("welcomeLabel.ImeMo
de")));
        this.welcomeLabel.Location =
((System.Drawing.Point)(resources.GetObject("welcomeLabel.Location")));
        this.welcomeLabel.Name = "welcomeLabel";
        this.welcomeLabel.RightToLeft =
((System.Windows.Forms.RightToLeft)(resources.GetObject("welcomeLabel.R
ightToLeft")));
        ...

```

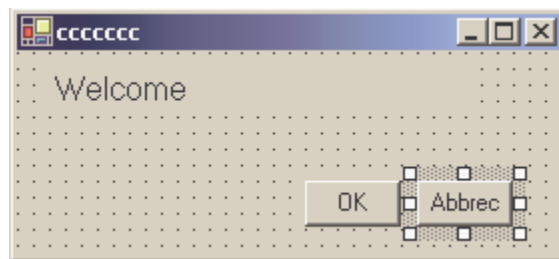
Note that the code has been modified so that the text for the label, button, etc. is obtained from the ResourceManager. In addition, even the properties like location are obtained from the ResourceManager. As a result of this, we may reposition the buttons or modify their size for different locales if desired. Continuing with the example, going back to the properties of the form, let's modify the language property to German:



Now notice in the solutions explorer that a resource file specific to German (for which the code is de) has been created as shown below:



Right now that file does not have any valuable information. Now, go to the design view of the form and modify the button text as shown here:



As you can see, the size of the cancel button is not big enough to fit the word “Abbrechen.” Now, we will reposition and resize the cancel button so the word fits as shown below:



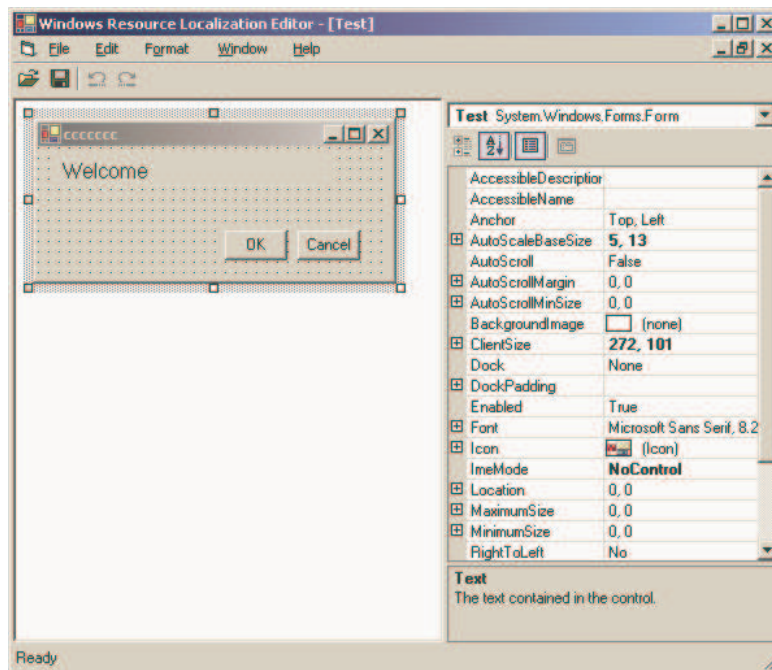
We may also modify the other texts including the title of the page, but for that we need to find some one who really knows German☺. Now take a look at the Test.de.resx file and you will see the new text we entered plus the details on the position of the buttons as shown below:

Data for data					
	name	value	comment	type	mimetype
▶	CancelButton.Location	192, 64	(null)	System.Drawi	(null)
	CancelButton.Size	72, 23	(null)	System.Drawi	(null)
	CancelButton.Text	Abbrechen	(null)	(null)	(null)
	OKButton.Location	136, 64	(null)	System.Drawi	(null)
	\$this.ImeMode	Inherit	(null)	System.Wind	(null)
*					

Now, you may switch the Language property of the Form to Default and you will see that the buttons go back to the default position and the cancel buttons text turns to “Cancel.”

As can be seen, it not only allows us to modify the text messages, it also allows us to reposition the text and to modify colors, images, etc, just about any thing we want to override or vary.

This of course leads to one issue. As seen above, as a developer, I can't really be entering the text and positioning for different languages. We need to find some one who knows German. Looking around I actually can find a few Germans! However, they may or may not be programmers. In order to make changes for a language, if we need to find a person who is proficient in that language and is also a programmer in that language is not reasonable. Further, I do not want to own that many licenses for Visual Studio, one for each person dealing with different languages. This is where the tool winres.exe (Windows Forms Resource Editor) comes in. But before we proceed further to discuss about winres, we need to understand one major issue. The resource file may be managed using Visual Studio or using winres, but not both! The format of the files used by the two tools is not the same. If non-programmers or third party will be assisting with the localization, then it is better to use the winres.exe instead of Visual Studio. In order to do this, build your application without any localization. Then set the localization property to true. This generates the default resx file (Test.resx in the example we discussed above). Hand off this default resx file to the language expert and he/she can use winres.exe to localize your application. An example of using winres.exe on Test.resx is shown below:



Clicking on File | Save gives us the following menu:



I selected German and clicked on OK. This created a file named Test.de.resx. One may now modify the text, move controls around and do what ever is necessary to localize this form for the selected language. When done, the locale specific resx file can be handed off to the developer to integrate in the product by creating a satellite assembly.

Challenges

In spite of these facilities, localizing an application is still a challenge. For the success of a product, one may have to understand a culture. This is not an easy task. Most of us are very familiar with our own culture and perceive things based on that. I read the following story some where: A soft drink maker wanted to boost sales in the Middle East. They decided to advertise using a poster. The poster showed a man who looked tired. The next picture showed him drinking the product. The third one showed him looking fresh and handsome. When they carried out the advertising campaign, the sales went down. Only then they realized that they read from right to left in that part of the world. In some cultures, certain colors and sounds are considered offensive. While the tools and techniques that we have discussed in this article come in handy, programmers experienced in localization would warn us that it takes more than that to be successful in localization.

Conclusion

The .NET framework and Visual Studio comes with a number of facilities and classes to help us localize an application. The use of satellite assemblies, resource files and resource manager makes localization easier. In addition, the tools like Winres.exe may prove to be instrumental to specific locale aware non-programming employees to help with the localization of your software application. For further details, please refer to the MSDN online reference.