Localizing Delphi and C++Builder applications

Sisulizer

9 November 2011

Introduction

This document describes the steps that you should perform when localizing Delphi or C++Builder applications. Localization means translating your application into another language. In the other words it means translation of user interface elements, resource strings, images and help files. For example if you have written your original application and help files in English and you want to start selling your application in Germany you have to translate your application and help files to German. Before you can start translating your application it must be internationalized. This means preparing your source code and user interface such way that it does not contain any language depend data such as hard coded strings. Internationalization is very important because it is necessary for localization and if it is not properly done then localization will be slower and more expensive.

The first half of the document deals with internationalization and common concept of localization. The second half is Sisulizer specific and it shows the localization process and explains how to take VCL's or FireMonkey's resource DLLs in use and finally it shows how to perform runtime language switch.

The first question most developers face when starting to internationalize a VCL or a FireMonkey project is *What Delphi or C++Builder should I use?* If possible you should use Delphi/C++Builder 2009 or later. They have far superior features to write international code to any earlier Delphi version. The reason is that Delphi 2009 and later creates Unicode applications unlike earlier versions that create ANSI applications. The biggest difference between these two is that in a Unicode application all strings by default are Unicode strings. This makes localization a bit easier, enables you to show international data (e.g. Japanese and Russian) on the same form, and it also makes it possible to run your application in Japanese on an English operating system. This is not possible with ANSI application unless to change the system code page of your computer and reboot it. A procedure that is way too restrictive and complicated for most needs. From localization point of view switching to Delphi 2009 or later is a good thing. However switching from pre-2009 to 2009 or later might require some additional work. In old VCL (before 2009) a string type as actually AnsiString. It is a string that uses variable length (1 or 2) bytes for each character and there are several different encodings, one for each character set. Each encoding is called a code page that contains table of characters the characters set contains. The basic property of a code page is that it contains a limited set of characters. The set covers only the characters of one language or language group. Western, Greek and Cyrillic code pages contain 256 characters each so each character is encoded as one byte. Asian code pages on the other hand contain thousands of ideographs (characters) so each character is encoded using one or two bytes. When you run an ANSI application the code page the application uses and the system code page of your operating system must match. A common misunderstanding is that in order to localize your application to Asian language you have to make your application a Unicode application. This is not true. You can perfectly localize an English or German ANSI application to Chinese but you need to run the application on Chinese OS or OS that has Chinese system code page set active. If you run the application on English system you all text will show up as mojibake.

There is another issue that strongly favors for Delphi 2009 or later. As told above Delphi has two kinds of string types: ANSI and Unicode strings. They are not compatible between each other because ANSI string can always contain only subset of Unicode characters. Pascal is strong typing language and if you try to assign a variable to another and the types of the variables do not match there is usually a compiler time error or at least warming. For some reason Delphi 2007 or earlier did not have this warning when

converting between different string types. Instead they performed an automatic conversion from Unicode to ANSI. Take a look at the following code.

```
function Sample(const value: WideString): WideString;
var
  str: AnsiString;
begin
  str := 'Sample';
  Result := str + value;
end;
```

The Sample function gets one Unicode string parameter and the function returns a Unicode string. The problem is that *str* variable in the code is defined as AnsiString so the string can only contain code page encoded ANSI strings. When the function adds *value* to *str* variable, an old Delphi compiler automatically converts Unicode string to an ANSI string, then adds the strings together and finally converts the new string from ANSI to Unicode. If the passed Unicode string (value) contains characters that the current code page of the system does not support data will be lost. This was very dangerous and very difficult to find. Fortunately Delphi 2009 and later have a compile time warning whenever string types do not match.

```
[DCC Warning] Unit1.pas(12): W1057 Implicit string cast from 'AnsiString'
to 'WideString'
```

This feature alone is worth buying Delphi 2009 or later.

This document is written for Delphi 2009 and later. If you use C++Builder 2009 or later you can perform almost the same steps. Basically the only difference is that C++Builder does not use *resourcestring* block inside the code but you have to use old fashion resource strings. If you use an older Delphi you can't perform all the steps and you have to use WideString type whenever this document uses UnicodeString type. We really recommend that you upgrade your compiler version to version 2009 or later. If you use Sisulizer for localization you won't need Delphi Enterprise but any Delphi edition is good.

Process

When Delphi compiler compiles an application it creates Windows PE file. That is a file that Windows operating system can run. The file extension is mostly .exe or .dll but other extension such as .ocx, .bpl or .cpl are also used. Common thing for all these files are that they contain two kinds of data: compiled code and resources. After you have compiled the code you should not change the compiled code. However it is possible to read and write the resource data. This is what localization tools do. They read the original resource data and they write translated resource data. There are several kinds of resource data such as form, string, bitmap. The form and string resources are the most important ones. In most cases you only have to localize them.

The simplest way to localize a PE file is to make copy of the original file and then translate the resource data in the file. As a result you will get one PE files for each language. For example if you original file is Project1.exe and it uses English language on forms and strings, and you want to create German version you localization tool creates German file (de\Project1.exe or Project1DE.exe). The actual file name or path is not important here but the fact that the German file must be another file and thus cannot have the same name. When you deploy you application the setup application either installs English or German application file. Another solution is to create two different setup applications: one for English and one for German.

A PE file can contain the same resource in several languages. This means that we can create a PE file that supports several languages. In that case all form and string resources are in two or more languages. The advantage of multilingual PE file is that you can always deploy the same file no matter that is the language of your user. The disadvantage is that because the PE files contains same resource in several time the size of the file is greater than original monolingual PE file. However the size difference is not that big. Delphi compiler cannot create multilingual PE files. Only few localization tools support them and ever fewer support multilingual Delphi files.

Whenever an application code is reading a resource data it looks the resource from its own application file (e.g. EXE). In addition of this VCL and FireMonkey have a build in feature where it can read the resources from a special DLL file that contains only resource and no code. For example if our application is Project1.exe the German resource DLLs is Project1.DE and Japanese resource DLLs is Project1.JP. Using resource DLLs bring one significant advantage: possibility to choose the language on run time. This makes is possible to always deploy the original EXE and in addition of that zero or more resource DLLs depending on your needs. In fact VCL and FireMonkey always look for resource DLL matching for the system locale of the computer when loading resource data. This means that if you run your application on Finnish Windows and there is a Finnish resource DLL (.FI or .FIN) the application uses resource of that DLL instead of the EXE. You can force VCL or FireMonkey to load any resource DLL by writing the language code to system registry.

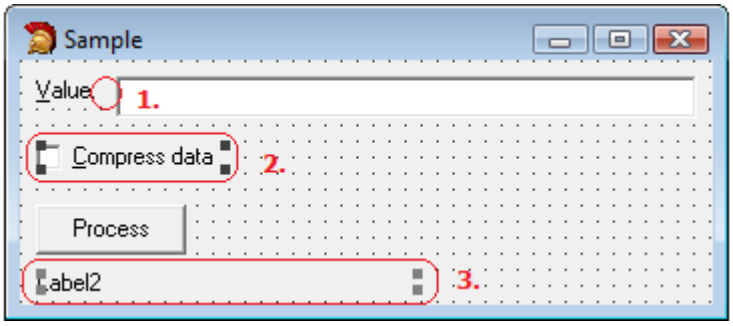
A neat thing of Delphi compiler is that it can create EXE files that contain all code including library code. These EXE files do not require any DLL or OCX in order to run. It makes installation and deployment much easier because you only have to copy the EXE file. Resource DLLs break this by bringing one DLL for each language. On the other hand resource DLLs make it possible to choose the initial language and with help of some code they also make it possible to change the language on run time. To make this possible with single file deployment it is possible to store the resource DLLs a custom resources inside the EXE. When application starts for first time it extracts the custom resource data to and creates the resource DLLs files. This process is called embedding resource DLLs. It brings your all the good features of above localization

methods such as single file deployment, possibility to choose the initial language, and even possibility to change the language on run time.

As described above there are four different ways to localize a Delphi application. It depends on your needs and on your localization tool that you want to use. Most localization tool support only localized PE files and resource DLLs. Very few support multilingual or embedded resource DLLs. Sisulizer supports all these methods.

User interface

Let's start with a sample. VCL\Tutorial\Original sample directory contains a project that requires internationalization. The project is very simple and does not do anything meaningful but the project shows most of the internationalization issues. The following picture contains TForm1 form that needs preparing for localization.



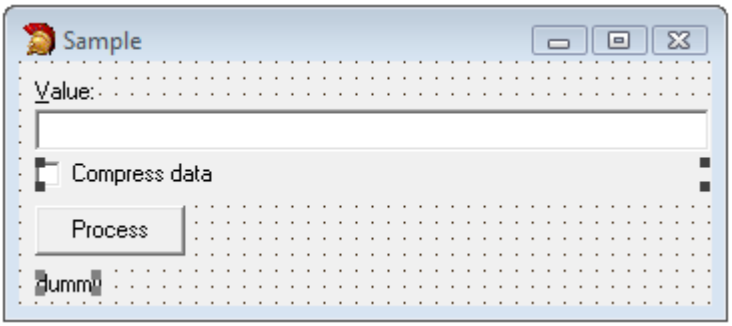
The form contains three cases where we have to change it in order localize the form easier. Each case is marked with red bounds and a number. Let's go through each step.

In the case #1 another control is following a control. The Value label is immediately followed by an edit box. This will cause you problems because it is very likely that the translation of the Value to other language will be longer even much longer that the original value. This will make label and edit to overlap each other. The translator can move the edit control more right to make room for the longer label but this will take some time and will cost. Remember if you localize to 10 different languages most likely every single translator has to do the same modification that will take a lot of time and even more important make your UI to look different on each language. A better approach is to design the original UI such way the translators need hardly ever relocate controls. In this case we can place the label and edit on different lines - label above the edit. This gives label possibility to expand very much without overlapping the edit control.

In the case #2 control's width is too short. TCheckBox does not have AutoSize property. You have to manually set the width of all check boxes to the maximum width allowed by its position.

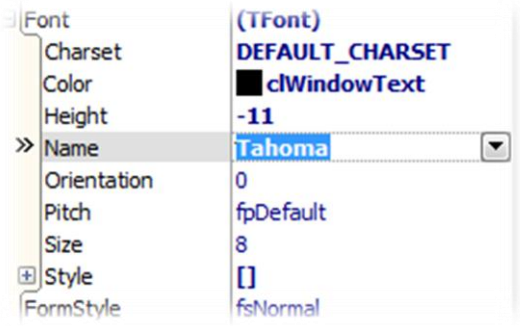
In the case #3 AutoSize property is not set true. Turn on the AutoSize property of TLabel. This makes sure that longer translation of the Caption property is not cut when drawn on the form. Label2 control needs another fix. Label2.Caption is set to Label2 on design time. On run time this is replaced with "Click the above button to process data" string. So the original string is not used at all. Keeping it will only make your localization project larger and more expensive. A good practice is to set all these Caption property some fixed value such as "dummy". After creating Sisulizer project you can easily exclude all strings having "dummy" value.

The following picture contains the same sample form after it has been prepared for localization. Label and edit are not on the same line any more. Check box width has been set to maximum possible width. Label's auto sizing has been turned on. Unused strings have been replaced with *dummy* word.



Fonts

The default font of Delphi forms is either Tahoma (Delphi 2006 and later) or MS Sans Serif (up to Delphi 7). MS Sans Serif looks a bit old fashion so it is better to use Tahoma. Keep the font name to Tahoma. This will ensure that Windows will replace it on fly to the most appropriate font if Tahoma does not directly support the script that is used. Tahoma supports directly on most scripts. However if string contains Asian characters then Windows will use the default font of the script. On Simplified Chinese it is Simsun. Instead of Tahoma you can use generic font called MS Shell Dlg 2. Windows maps it to Tahoma. Another issue is the font size. The default font size on Western Windows is 8. However on Asian Windows it is 9. However if you use Tahoma as the font name Windows will automatically use increased font size on Asian languages.



Using Tahoma (or MS Shell Dlg 2) always ensures that the font of your application uses the default user interface font of the target operating system. You do not have to localize any fonts on your application. This will speed up the localization process and make it cheaper.

Hard Coded Strings

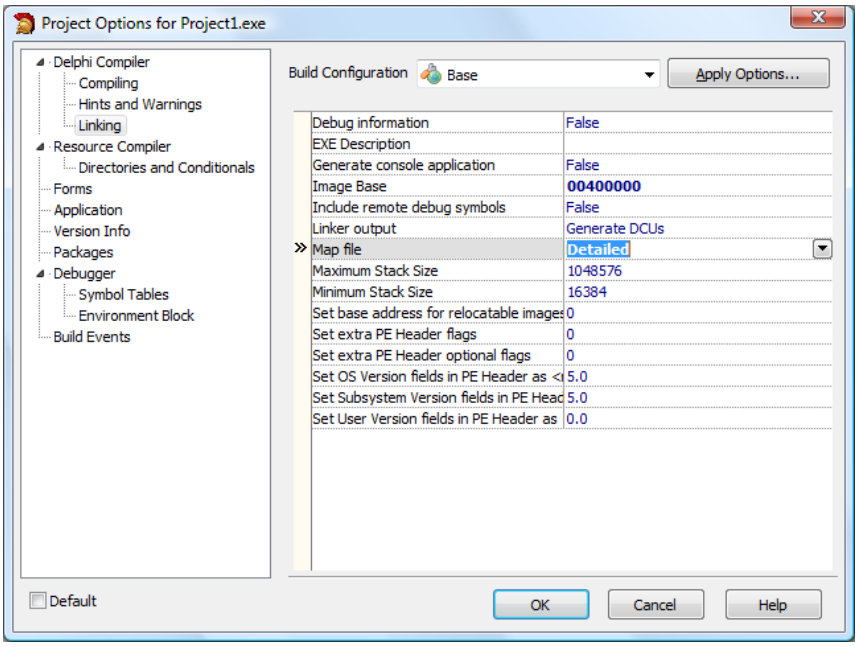
Most applications have hard coded strings inside the code. Some of them must not be translated. Some of them should be translated. If you want to localize such a string you have to remove them and replace them by resource strings. This is called resourcing. Fortunately it is very easy process in Delphi. Take a look at the following code. It contains a hard coded string that is used to set Caption property of label.

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  Label2.Caption := 'Click the above button to process data';
end;
```

To resource this string add a *resourcestring* block above the begin block. Give a unique name for the resource string id and set the resource string to match the hard coded string value. The name of the resource string should also be as describable as possible. SClickButton is much better than SStr1. Finally replace the hard coded string with the resource string.

```
procedure TForm1.FormCreate(Sender: TObject);
resourcestring
  SClickButton = 'Click the above button to process data';
begin
  Label2.Caption := SClickButton;
end;
```

When Delphi compiler compiles a resource string it stores the string as a standard Windows string resource and assigns an id for the string. If you add new resource strings into the application or delete existing ones, the compiler will give most resource strings new ids. This will most likely cause loss of translations or existing translations to be replaced with wrong translations: a situation that you want to avoid. To prevent this let Delphi to create Delphi resource string file (.drc) file and assign the file name so Sisulizer can use it to link resource string names to ids (e.g. SSampleString equals 4567). The resource string name won't change unless you intentionally change it. This makes resource string name much better context value as resource string id. DRC files use the same name as the project file but have .drc file extension (e.g. C:\Samples\Project1.drc). To create a .drc file choose Project | Options menu from Delphi, select Delphi Compiler –Linking and set Map file to Detailed. Whenever you recompile your project Delphi will create a new DRC file that contains updated resource string names and ids.



Once you give a resource string a name (e.g. SClickButton) do not change it. You can safely change if before localization process has been started but after the project has been sent for localization no id, resource string, component or form name, should be changed any more. Most localization tool loose existing translations if you change the value of id. Sisulizer looses translation only if you change both id and value at the same time.

You can localize a Delphi binary file without specifying the DRC file but in that case Sisulizer uses the resource string ids as the context. It is very likely that Delphi compiler will change the resource strings ids next time you compile your project. This will cause lost of translations or switching of translations. This is why it is very much recommended to specify a DRC file.

C++Builder does not use DRC files. C++Builder uses traditional STRINGTABLE resources of RC files to store resource strings and .h files to specify the resource string ids

VCL itself contains hundreds of message strings. They are added to the resource string resources of your application just like your own resource strings. However if you use runtime packages VCL's resource strings are not linked to your application but are inside the package files (.bpl). If you want to localize them you have two choices. The first option is not to use runtime packages which case the string are linked to the application. In this option VCL’s resource strings and possible form are include to the same resource data along with your own resource strings and forms. The second option is to localize the runtime package files. They are Windows DLLs and are localized in the same way as you application files (.exe). Just add them to your Sisulizer project. We recommend the first option where VCL units are linked into the application file. It makes localization, deploying and possible runtime language switch a lot easier.

If you use Sisulizer to localize you application you can also set comments and maximum lengths for the resource string directly in your source code. See Comments and other additional information on page 27.

Code enabling

Now we have removed hard coded strings. The next step is to modify the rest of the code such way that it works on every country and it does not have any language depend code. In most cases you have to check two kind of code: dynamic messages and conversion code.

Dynamic messages are strings that are created on runtime combining static text with dynamic parameters. Most often they are messages for the users but can also be text on dialogs, output files or reports. These messages are called dynamic because you do not know that actual text on compile time. Instead application uses some code to create the message on run time. Take a look at the following code where application creates a message and show it. GetName returns the name of the object and it is added to the static text of “Computer name is “.

```
procedure TEnableForm.Button1Click(Sender: TObject);
begin
  ShowMessage('Computer name is ' + GetName);
end;
```

If GetText returns “SAMPLE” the message will be “Computer name is SAMPLE”. How can we localize this such way that the message will be given in the same language as the user interface. The first idea might be to take resource string in use.

```
procedure TEnableForm.Button1Click(Sender: TObject);
resourcestring
  SMsg = 'Computer name is';
begin
  ShowMessage(SMsg + GetName);
end;
```

This seems right but it still has a limitation that might prevent proper localization. The reason is that the ode assumes that the message starts with “Computer name is “ and then continues with the item name. This has to drawbacks. First is the fact that in some other language but English the word order might be different. For example it might be SAMPLE is the computer name. So the message starts with the item name and then continues with “ is the computer name”. Using the above code this can’t be archived but the message must always end with the item name. Another flaw is that the resource string in partial: it only contains “Computer name is “. The sentence is not complete and it might be difficult for translator to translate. A better approach is to use message patterns. A message pattern is a string that contains one or more parameter placeholders. The code combines the pattern and the parameters on run time to create the message. Because pattern is the full sentence it is easier to translate and the translator can move the placeholders to any location to match the grammar need of the target language. Delphi contains Format message that uses two parameters: the pattern string and an array or parameters. Our final code version uses Format function.

```
procedure TEnableForm.Button1Click(Sender: TObject);
resourcestring
  SMsg = 'Computer name is %s';
begin
  ShowMessage(Format(SMsg, [GetName]));
end;
```

If the translator wants to start the message with the name he or she can translate the pattern to “%s is the computer name”. A pattern can contain multiple parameters. For example “%s file was created by %s”. The code could be like this:

```
Format('%s file was created by %s', [fileName, userName]);
```

If translator wants to swap the order of the parameters he or she can add the indexes. For example Finnish translation would be “%1:s loi %0:s-tiedoston”. Here the user name is before file name. The parameters must be in reversed order and this is why they contain indexes (%**1:**s). If pattern does not contain indexes they are automatically ordered starting from left. “%s file was created by %s” is same as “%0:s file was created by %1:s”.

Most applications show numbers and dates on user interface or reports. A code enabling is needed here too. The reason is that there are several different ways to format number, currencies and dates. American way to show date is mm/dd/yy where first is month following by day and year each in two digits and separated by a slash. For example 12/24/08 is Christmas Eve 2008. However the same date in Finland would be 24.12.2008. The format is dd.mm.yyyy. There are several other formatting styles used in other countries. If you have code that always uses American date format you have to enable the code. The following sample only works in USA (and countries using the same data formatting).

```
procedure TEnableForm.FormCreate(Sender: TObject);
var
  year, month, day: Word;
begin
  DecodeDate(Now, year, month, day);
  Label1.Caption := 'Today is ' + IntToStr(month) + '/' + IntToStr(day) +
'/' + IntToStr(year);
end;
```

Use FormatDataTime functions instead. The functions can format the give date to show, long or customized string. The function always uses the formatting rules of the active locale (language + country).

```
procedure TEnableForm.FormCreate(Sender: TObject);
begin
  Label1.Caption := Format('Today is %s', [FormatDateTime('ddddd',
Now)]);
end;
```

Change you code that converts numbers or currency value to string.

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  Label1.Caption := 'Bike is $' + FloatToStr(499.90);
end;
```

Above code assumes that currency is $ and word order is “Bike is “ following by the value. To make the code world ready use resource strings and Format function.

```
procedure TForm1.FormCreate(Sender: TObject);
resourcestring
  SMsg = 'Bike is %m';
var
  c: Currency;
begin
  c = 499.90
  Label1.Caption := Format(SMsg, [c]);
end;
```

VCL contains other function but Format to convert currencies to string. System unit contains CurrencyToStrF and FormatCurr functions that you can also use.

Whenever you create strings on run time by using + to combine strings be careful and think if each parts of the string should be combined to one message pattern that contains one or more parameters.

Common dialogs

The Dialog sheet in Delphi’s Tool Palette contains several dialog components. These are so called common dialogs. TOpenDialog for example shows the Open dialog that is used to select a file. Common for all these components is that they are not 100% VCL components but wraps around dialog controls of WIN32. The dialog code and user interface resource are not linked to compiled application. The code and resource are in a separate DLL that belongs to Windows. When you run a Delphi-compiled application that uses the dialog component the language of the dialogs depends on your operating system. Let’s have an example. You have created an application on English OS using English Delphi. Your application is 100% in English. When you run it the open dialog appears in English. If you localize you application into German Sisulizer can fully localize all elements into German. However the open dialog is not part of your application. The open dialog DLL is always there because it is part of OS but the language of the DLL depends on the language of the operating system. On your English OS the language is English. This is why the open dialog always appears in English on your computer even if you run your localized (German) application. The case is different on your German client. If he runs your original English application it appears in English but the open dialog appears in German. If he run the German version then application itself and the open dialog appears in German. This is why should not worry even you can not translate the common dialogs. The end user of your application has dialog DLLs that match his or her language.

TDBGrid

When you use TDBGrid you have two options. Either you populate TDBGrid.Columns manually on design time or you leave it empty and VCL populates the column on run time to default columns.

When you use design time columns approach you use Delphi IDE to add the columns that you want to be visible. Each TColumn contains Title.Caption property that specifies the caption that is shown on the grid. Unfortunately VCL does not save the Caption property into DFM if the value of the Caption equals to the

name of the field in the database. If the field names are plain English and even if you add columns manually to TDBGrid the caption values are not saved into DFM. The following code contains one such a column item of DFM.

```
  Columns = <
    item
      Expanded = False
      FieldName = 'Id'
      Width = 30
      Visible = True
    end
    ...
```

There is no Caption property here. To localize captions you have two options. Either you set the Title.Caption value such that it does not match the field name. For example if field is “ID” you assign “Id”. This way VCL saves it and it can be automatically localized. Another way is to assign the Caption properties on run time using resource strings. The best place for that is the TForm.OnCreate event.

```
procedure TMainForm.FormCreate(Sender: TObject);

  procedure SetCaption(index: Integer; const caption: String);
  begin
    Grid.Columns[index].Title.Caption := caption;
  end;

resourcestring
  SId = 'Id';
  SName = 'Name';
  SPopulation = 'Population';
  SCapital = 'Capital';
  SDescription = 'Description';
begin
  SetCaption(0, SId);
  SetCaption(1, SName);
  SetCaption(2, SPopulation);
  SetCaption(3, SCapital);
  SetCaption(4, SDescription);
end;
```

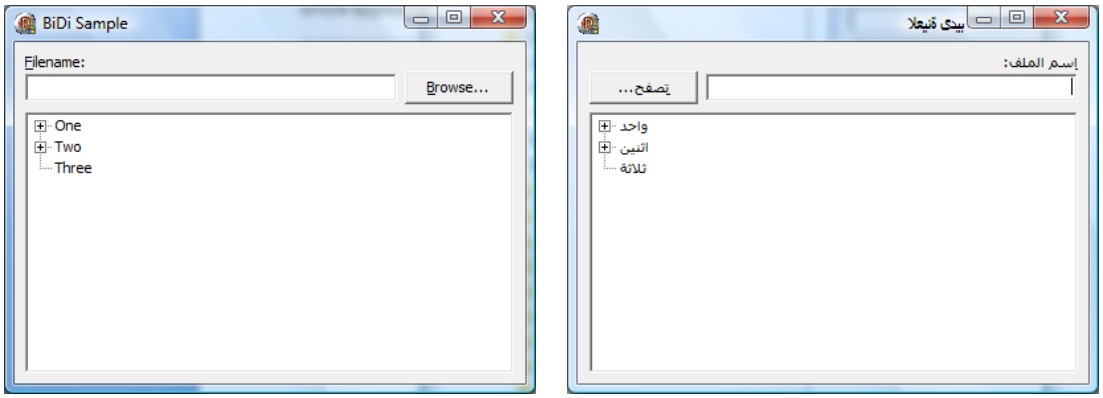
When you use default columns you don't populate the columns and this makes VCL to automatically populate it on run time. There is no Columns data in DFM so they cannot automatically be localized. You have to assign the Caption properties on run time using the same code as in the above sample.

See <sisulizer-data-dir>\VCL\Delphi\DBGrid for sample how to localize TDBGrid.

How to support bi-directional languages

Arabic, Hebrew and Persian are so called bi-directional languages. They differ from other languages such way that text is read from right to left. This also means that user interface must be mirrored compared to Western user interface. This makes localization to bi-directional languages more difficult. Fortunately VCL contains several properties and function to control both reading order and to mirror forms. The most important property is TApplication.BiDiMode. It specifies the default reading order of the application. Each control contains two properties BiDiMode and ParentBiDiMode. If ParentBiDiMode is True the controls

uses the BiDiMode of its parent. If there is not parent like forms have then Application.BiDiMode is used. By default ParentBiDiMode is true so by changing Application.BiDiMode is enough to set the reading order of whole application. Setting the reading order is not enough; you also have to mirror the user interface.



Some localization tools can change BiDiMode to match the target language. Some tools can even mirror the form layout automatically when creating bi-directional data. However there are two problems here. VCL optimizes DFM form such way that if the property value is default it is not written. By default BiDiMode is bdLeftToRight and is not written to DFM file. When a localization tool read DFM data it does not find the property and this is why it does not let to change it. Because ParentBiDiMore is True by default it is enough to set only top level BiDiMore to true. This top level BiDiMode is in TApplication and it does not have a resource. Localization tool that works only with resources cannot change the value. It must be done manually in code. TWinControl contains FlipChildren method that mirrors the controls and optionally all its child controls. This cannot be controlled by a property but it has to be done in code. To mirror a form just call the method in the OnCreate event.

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  Form1.FlipChildren(True);
end;
```

This is preferable over localization tool mirroring because in many cases you move and size components on run time. The DFM file contains unmoved and unsized layout and if localization tool mirror is your layout code gets a lot more difficult. This is why it is better to first layout in the OnCreate or OnShow events and after that call FlipChildren.

Of course in a real world application you want to change the reading order and mirror the forms only if you run application in bi-directional language. To do that you have to check it the active locale is bi-directional.

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  if IsActiveLocaleBidi then
  begin
    Application.BiDiMode := bdRightToLeft;
    Form1.FlipChildren(True);
  end;
end;
```

Active locale tells the language id of the resource that the application is currently using. VCL does not contain IsActiveLocaleBidi function. It comes with Sisulizer units. The function checks what is the language

of the localized application (if running localized EXE), what is the language of the loaded resource DLL (if using resource DLLs), or what is the active language of multilingual EXE (if using multilingual EXEs).

See <sisulizer-data-dir>\VCL\Delphi\BiDi for samples how to localize to bi-directional languages.

Use frames and inherited forms

When you write properly internationalized code you have to change the default behaviors of each form and frame. This is why you better derive an abstract form from TForm and derived all your forms from this form. Do same for frames. This makes it easy to initialize the form in the constructor of the base form.

Use inherited forms especially if you localize you application into bi-directional language. As told in the previous chapter you have to mirror each form on OnCreate or OnForm event. If you use form inheritance you have to perform this mirroring only in the base form.

FireMonkey does not support frames so you can use frames only if you use VCL.

Converting your project to Delphi 2009 or later

This document is about localization, not about converting projects to Delphi 2009 or later. However I have found few things useful when converting pre-2009 application to 2009 or later.

The most important thing is to keep all warning turned on and do not ignore string case or data loss warnings. Change you code until there are no warnings.

Many applications use string to store binary data. Strings are not meant for it but it provides more convenient way to handle binary buffer as AllocMem and pointers. No need to allocate memory or free memory afterwards. Just have to set the length of the string before read data. Let's take a look at a sample.

```
procedure ReadFile(const fileName: String);
var
  buffer: String;
  stream: TFileStream;
begin
  stream := TFileStream.Create(fileName, fmOpenRead);
  try
    SetLength(buffer, 100);
    stream.Read(buffer[1], 100);
  finally
    stream.Free;
  end;
end;
```

The code read the first 100 bytes from a file and stores it to a buffer. Buffer is String. Code sets the size of the string to 100 and then read the data into buffer. This worked very well in pre-2009 and was efficient code too. This was because String as AnsiString in pre-2009 and every character was one byte. In 2009 String is UnicodeString and every character is two bytes. This is why you have to change the type of buffer from String to AnsiString or RawByteString.

```
procedure ReadFile(const fileName: String);
var
  buffer: AnsiString;
..
```

Pre-2009 has only one type of Ansi string called AnsiString. 2009 has several each having different code page attached to string. If there is a string operation where two different string types are involved 2009 actually converts both string to Unicode before operation and then converts result back to Ansi.

The same thing occurs when you call a function that takes Ansi string as parameter but the passed parameter is not the same type. In that case 2009 converts passed strings to Unicode and then to the Ansi string of the function.

Pre-2009's Unicode string type was WideString. Of course it is still there but 2009 contains another Unicode string called UnicodeString. It is more efficient and reference counted just like AnsiString. You should use UnicodeString as a primary Unicode string type. Use WideString only if you pass a value to COM function that requires BSTR typed variable.

This document is not about converting old Delphi application to Delphi 2009 but about localization. There is an excellent white paper about Delphi 2009 and Unicode by Marco Cantú

Internal data

Above paragraphs have described how you internationalize the user interface and messages of your application. The next step is to internationalize your data. If your application does not handle any string data or other data that is country depend such as postal addresses you do not have to change anything. However if your application deals with strings or lets user to enter text you have to make sure that it can handle different scripts.

The basis of internationalizing our data is to use Unicode. The default string type in Delphi 2009 is UnicodeString that contains UTF-16 string. It can contain all possible characters and ideographs used around the world. You do not have to do anything special. Just use string or UnicodeString whenever you handle strings. You can assume that every character takes two bytes and you can index strings linearly. There is one catch here. Unicode space is huge and it can contain several hundredths of thousands of characters. The first 64 000 characters are the basic Unicode characters. It contains all the characters except rare Chinese ideographs. In order to support this UTF-16 uses surrogate pairs where the characters above basic Unicode characters use four bytes instead of two. This makes linear accessing of string impossible because you have to access the string from beginning in order to calculate the index of desired character. You only have to do this if your application must support all Chinese characters. Most applications are fine with standard Chinese characters and there is no point to use surrogate pairs.

Most applications save the internal data to a permanent storage. Such storage is most often either file or database. VCL\Tutorial\File sample directory contains a project that stores all internal data in Unicode but reads and writes data in several different encoding including code page encoded data. The sample uses XML, text and binary file formats to store its data. Open the sample to Delphi 2009 and play with it.

Use XML files whenever it is possible. This is because the default character encoding of XML is UTF-8 that is Unicode. If your data contain mostly Asian characters you might what to use UTF-16 encoded XML. XML file format is very flexible and you can add new items to your elements without breaking the existing file. This means that you can modify XML format and still keep excellent backward compatibility. This is very hard if you use text or binary files. Reading and writing XML files from your code is simple. When you access XML files using you XML class everything is in Unicode strings and you do not have to convert strings at any way. However in some cases XML files can't be used. Either there is a speed issue or data must be in conventional text files. In these cases you might need to read and write non-Unicode data. This means that you have to convert your internal data to code page encoded data before writing and after reading you have to convert code page encoded data to Unicode. This might seem difficult be in actually is quite simple. You read the bytes from the file and sore them to an AnsiString typed variable. Then you convert the AnsiString to UnicodeString. You can use WIN32 API's *WideCharToMultiByte* and *MultiByteToWideChar* functions to perform conversions. However they are inconvenient to use because they require many parameters. Sisulizer has LaCommon.pas unit that contains two functions: *AnsiToUnicode* and *UnicodeToAnsi*. They both take only two parameters: a string and a code page to be used. Here is a sample code that reads a string from the stream. The code first reads the length of the string in bytes, then the actual bytes and finally converts the ANSI string to a Unicode string. Code assumes that data in the file is encoded using code page specified in CodePage property.

```
procedure ReadString: String;
var
  len: Word;
  ansi: RawByteString;
begin
  len := ReadWord;
  SetLength(ansi, len);
  stream.Read(ansi[1], len);
  Result := AnsiToUnicode(ansi, CodePage);
end;
```

When you write a file you first convert UnicodeString to ANSI using *LaCommon.UnicodeToAnsi* and then write the bytes of the AnsiString to the file.

```
procedure WriteString(const value: String);
var
  len: Word;
  ansi: RawByteString;
begin
  ansi := UnicodeToAnsi(value, CodePage);
  len := Length(ansi);
  WriteWord(len);
  stream.Write(ansi[1], len);
end;
```

When you write the file you need to convert data from UTF-16 (that is used in VCL's UnicodeString) to the target code page. There are dozens of code pages and there might be several code pages for single language. A single byte code page can only contain 256 characters. This is not enough for to contain all diacritical characters used in all language using Latin alphabets. This is why single code page can handle only certain group of languages. Typically there are four major code page groups in Europe: Western, Eastern, Greek and Cyrillic. For example most Western language such as English, German and French use Windows code page 1252. There is a very similar ISO 8859-1 code page that is used on Linux and many web pages. Macintosh uses Mac Roman code page. In addition there are several legacy code pages (e.g. DOS, EBCDIC) that are compatible to Western European languages. Windows XP and Vista support most existing code pages. Windows API has a code page number for each code page. When you are converting string data from ANSI to Unicode you need to know the code page number. If you want to enumerate the code pages you system supports use EnumSystemCodePages function. See *TForm1.FormCreate* event in our file sample to see how this function is used.

Japanese

Even Unicode has 100% support for Japanese there are still some other encodings that are very widely used in Japan. They are Shift JIS, ISO-2022-JP and EUC. If you write an application for Japanese markets you probably have to support them in addition of Unicode. Shift JIS, ISO-2022 and EUC are all code page encodings and cover approximately the same set of characters. Only the encoding and code points differ. Windows code page id for Shift JIS is 932, ISO-2022 is 50220 (50221 and 50222 are also used) and for EUC is 51932. You need to use MLang when converting data to/from EUC. The following table contains the most common encodings supporting Japanese.

Encoding	Code page	Description
UTF-8	65001	Unicode. Supports all characters.(1)
UTF-16	-	Unicode. Supports all characters. (2)
Shift JIS	932	Supports most Japanese characters.
ISO-2022-JP	50220, 50221 and 50222	Supports most Japanese characters.
Mac Japanese	10001	Supports most Japanese characters.
EUC	51932	Supports most Japanese characters. You need to use MLang interface to convert from/to Unicode.

(1) Windows has a code page id for UTF-8 and you can use UnicodeToAnsi and AnsiToUnicode functions to convert between UTF-16 and UTF-8. However it is more convenient to use Utf8Encode and Utf8ToString functions because they do not require any code page parameter.
(2) UTF-16 has two variants, one for little-endian (LE) and another for big-endian (BE) byte orders.

Chinese

Like Japanese also Chinese has several legacy encodings. What makes Chinese even more complicated is the fact that there exists two written Chinese: Simplified and Traditional. Simplified is used in People's Republic of China and Singapore. Traditional is used in Taiwan and Hong Kong. Unicode supports both Simplified and Traditional Chinese. However some rare Chinese characters do not fit to Basic Multilingual Plane of Unicode but are stored on Supplementary Ideographic Plane. To encode these characters UTF-16 uses surrogate pairs.

Encodings that are used in Simplified Chinese are GB 2312 and GB 18030. The later is a superset of GB2312. Both are very commonly used in China. In fact your application must support GB 18030 in order to be able to sell it in China. Windows XP and later support GB 18030. GB 18030 contains much more characters than those used in Simplified Chinese. Actually it contains whole Unicode so it is considered as one of Unicode encodings. The following table contains the most common encodings supporting Chinese.

Encoding	Code page	Description
UTF-8	65001	Unicode. Supports all characters.
UTF-16	-	Unicode. Supports all characters.
GB 18030	54936	Unicode. Supports all characters.
GB 2312	936	Supports most Simplified Chinese characters.
ISO-2022-CN	50227	Supports most Simplified Chinese characters.
Mac Chinese	10008	Supports most Simplified Chinese characters.
Big5	950	Supports most Traditional Chinese characters.
ISO-2022-TW	50229	Supports most Traditional Chinese characters.
Mac Traditional Chinese	10002	Supports most Traditional Chinese characters.

Korean

Like Japanese and Chinese Korean also has several legacy encodings. The following table contains the most common encodings supporting Korean.

Encoding	Code page	Description
UTF-8	65001	Unicode. Supports all characters.
UTF-16	-	Unicode. Supports all characters.
Windows Korean	949	Supports all Korean characters.
EUC-KR	51949	Supports all Korean characters.
ISO-2022-KR	50225	Supports all Korean characters.
Mac Korean	10003	Supports all Korean characters.

Databases

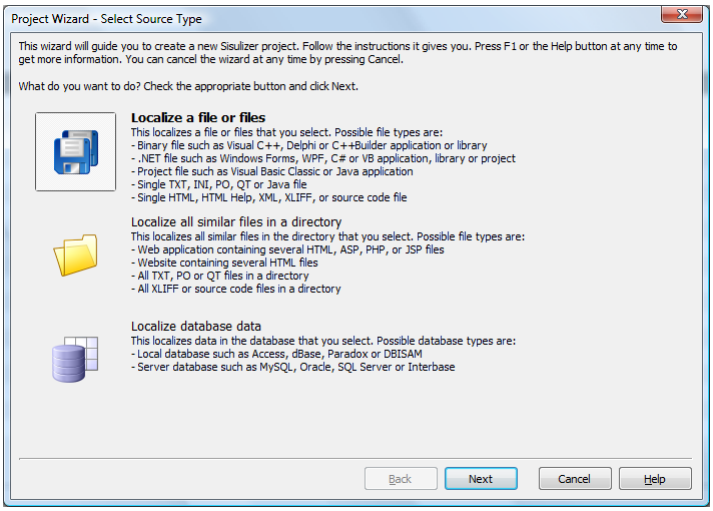
Most modern databases support Unicode fields. When you design your table structure make sure that you use Unicode fields instead of ANSI fields. If there is no Unicode field support in your database you have to use either UTF-8 or code page encoded ANSI strings. In both cases you have encode your Unicode string before you write them and decode them after you have read them. Using UTF-8 is recommended over code page because you do not need to know the code page where data is encoded with.

Using Sisulizer

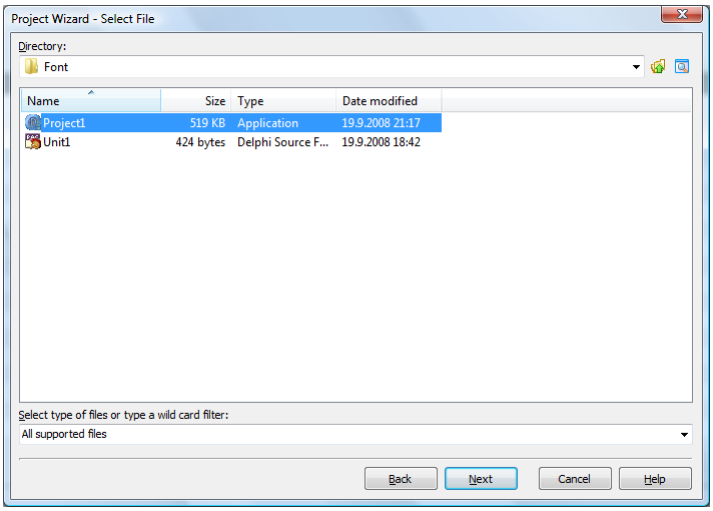
We have now internationalized our Delphi project. It is time to localize it. The rest of the document is used to describe how to use Sisulizer to localize a Delphi application. We will first perform a simple localization where the output files are actually localized EXE file. After that we will take resource DLLs in use. This will give you an option to choose what the language of the application is when it starts. The final step is to use Sisulizer units to make the application multilingual so it can start in one language and the user can change the language on run time.

Create project

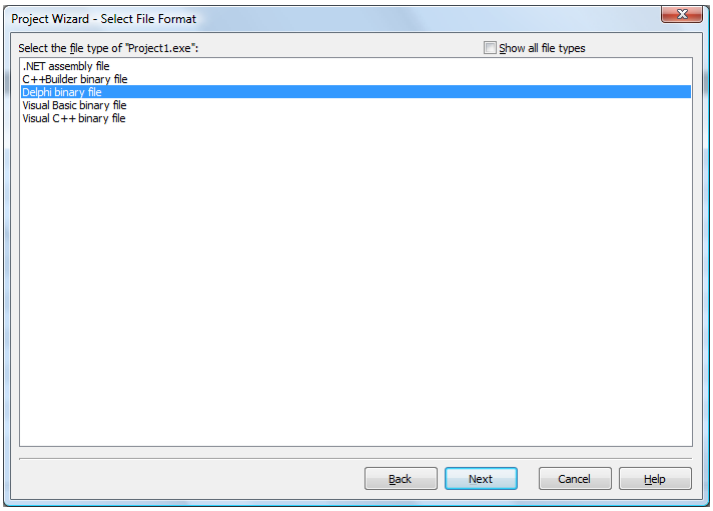
Make sure you have compiled your application. Start Sisulizer and choose File | New to start Project Wizard. Click *Locale a file or files* and click Next.



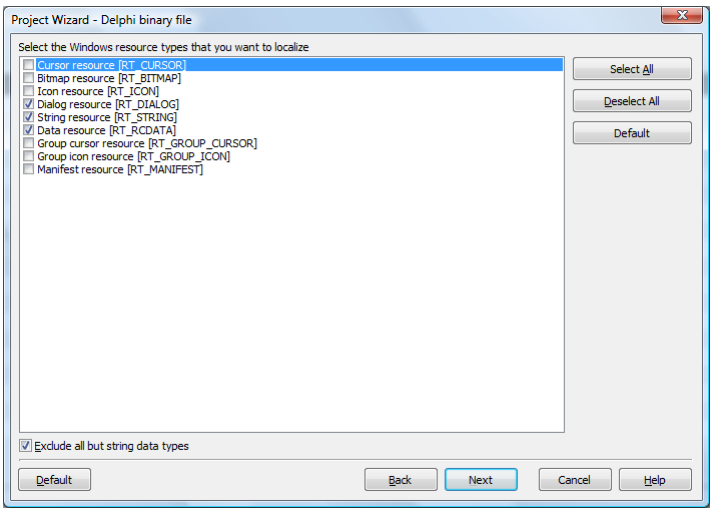
Select File page appears. Browse the directory where your application file (.exe) is located and select that file. Click Next.



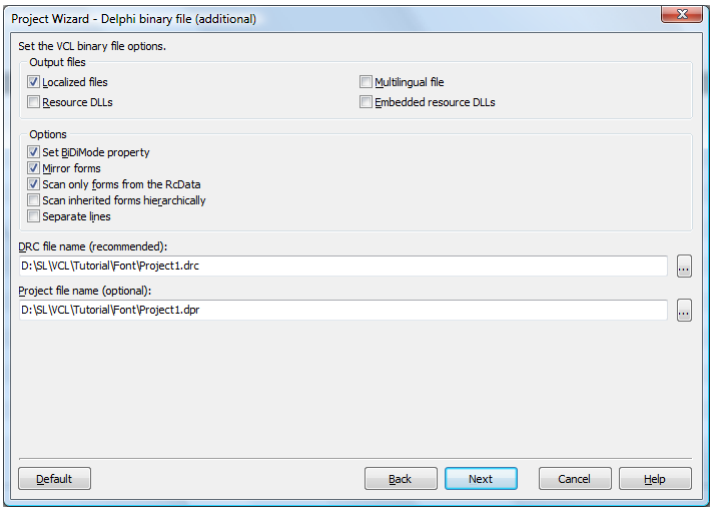
Select File Format page appears. By default Sisulizer detects the application file as a Delphi binary file. If it does not detect choose *Delphi binary file* in the list. Click Next.



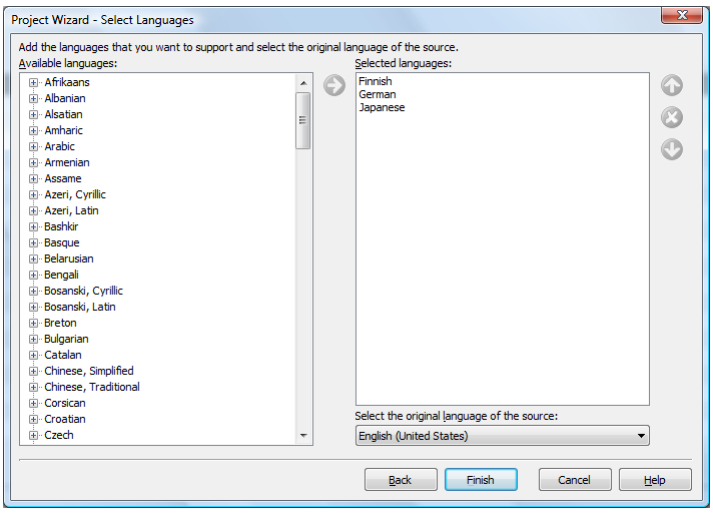
Delphi binary file page appears. This page shows the resource types that the application contains. Check the resource types that you want to localize. By default Sisulizer checks form, dialog and string resources. If you have other resource types that you want to localize check them. Click Next.



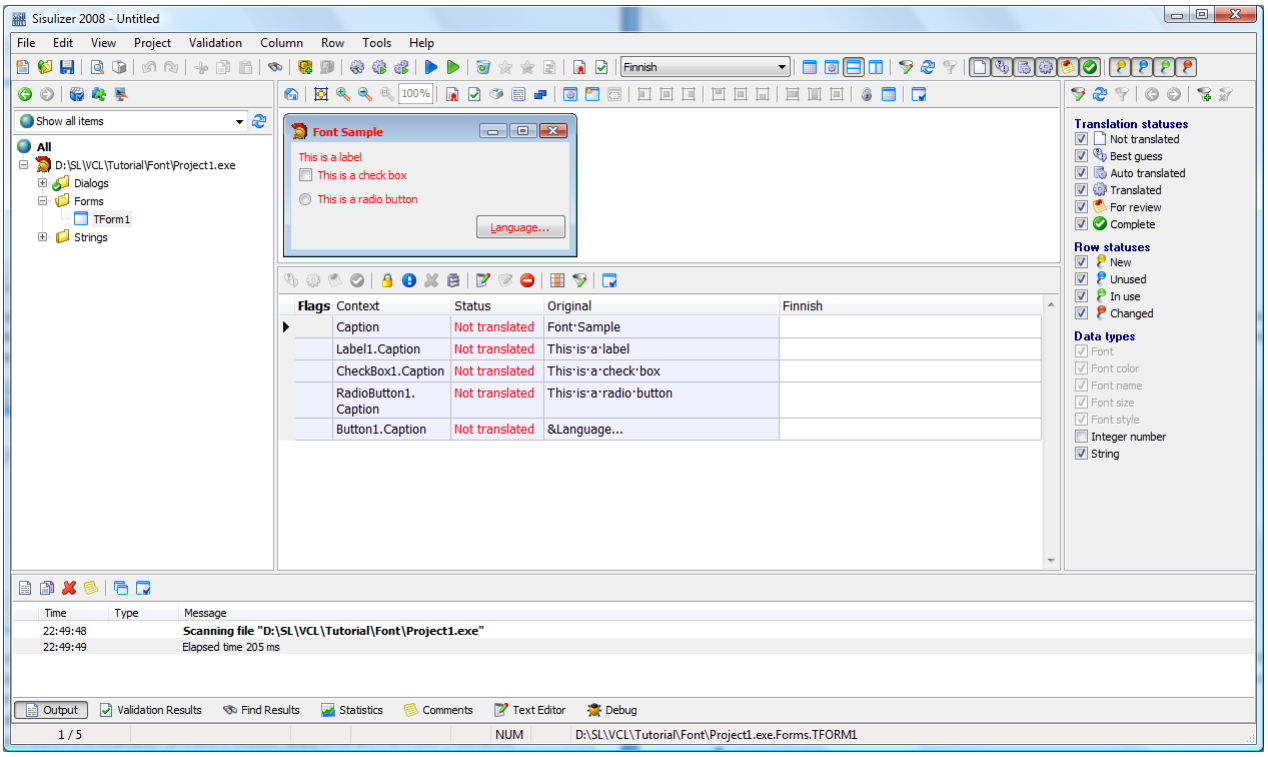
Delphi binary file (additional) page appears. This page contains Delphi related options. You can choose default file type(s). By default Sisulizer creates localized files. It means that Sisulizer create a separate application file for each target language(s). The application file is the same as the original application file but the resource data has been translated. By default Sisulizer detects DRC and project files. Both are recommended to be used in the localization process. If Sisulizer failed to detect them click … button to select a file. Click Next.



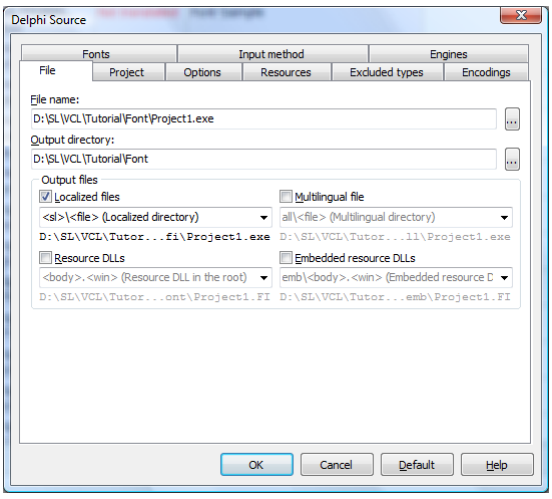
Select Languages page appears. Use this page to select the language of the original file (in most cases Sisulizer detects it) and to select the target language(s). You can later change the original language and add more target languages or remove existing ones.



Click Finish to finish the wizard. Sisulizer creates a new project, add the application file into the project and finally scans the project to find resource items from the application file.



Sisulizer project can contain any number of files. A file that has been added to the project file is called *source*. To add a new source right click All in the project tree and choose *Add Source*. You can configure the source by right clicking the file name in the project tree and choosing Properties. The following dialog is shown.



Use the dialog to set the localization options such as output directory, resources to be localized, etc. To get detailed information about the dialog select the sheet and click Help. Now we have created a project for our application. The next step is to translate it.

Translate

Sisulizer provides several ways to translate projects. You can translate manually by entering in the project sheet. The sheet works in the same way as Excel so if you are familiar to Excel you learn very quickly to use it. If you are not familiar to Excel it is easy to learn how to use the sheet. Just use mouse select a cell where you want to enter translation and start typing. You can also use arrow key to select a cell.

Another powerful way to translate projects is to import translations. Sisulizer can import translation from several sources such already localized files, glossary files (TMX, Excel, TXT, CSV) or from databases. You can import data to project by choosing File | Import. It starts Import Wizard that lets you to select the source and import options. You can also import to single language column by choosing Column | Import.

Sisulizer has build in support for translations memories. They are storages that keep existing translations. You can save existing translation from project to translation memory by choosing File | Save to Translation Memory. You can make translation memory to translate the project by choosing Project | Translate Using Translation Engine.

Final way to translate a project is to send it to translator. This is very easy in Sisulizer. Choose Project | Exchange. It lets you to create a file to be sent to translator. When translator has translated the file he or she will send it back to you and you can import translation from the file by choosing Project | Import.

You can find more information about translating the project from Sisulizer online help.

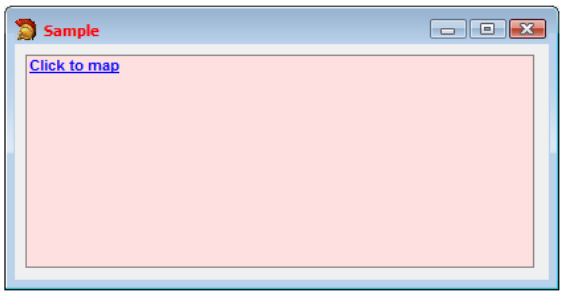
Build

Last step in the translation process is building localized files. Sisulizer takes care about this. It reads the original files merges the translations from the project file to create localized files. You have two ways to build the files. The first one is to open Sisulizer project and to choose Project | Build All. This makes Sisulizer to build localized files for all languages in the project file. If you want to build files for one language select the language and choose Project | Build Item. If you have Sisulizer Enterprise edition you can also use Sisulizer's command line tool, SlMake.exe, to build localized files. Go to Sisulizer directory and type SlMake on command line and press Enter to learn about SlMake. By using SlMake you can integrate Sisulizer into your make process.

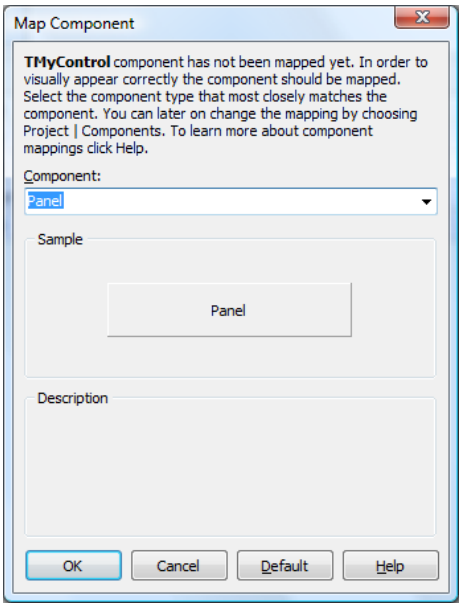
Now we have covered the basic Sisulizer localization process. The remaining chapters will cover some advanced topics about Sisulizer.

Component mapping

Visual editor is an essential part of Sisulizer. It is a WYSIWYG editor that shows the selected form visually above the translation sheet. Editor helps seeing the context of strings and makes it possible to array layout of the localized forms. In order to visually show components correctly Sisulizer needs to know what kind of component each component id. For example TLabel is a label component and TTreeView is a tree view component. Sisulizer has dozens of different build in visual components to be used in editor. The components cover most common component types such as labels, check boxes, grids and menus. Sisulizer uses component mapping to map the actual component name (e.g. TLabel) to Sisulizer's build in component type (e.g. label component). If you choose Tools | Platforms | VCL or Tools | Platforms | FireMonkey menu and select Components sheet you can see the existing component mapping. By default all VCL and most 3rd party components have been mapped. If you use your own components or some 3rd party component that have not yet been mapped Sisulizer can correctly scan and localized the component but can't correctly show it on visual editor. In that case it shows the component as pink box that has *Click to map* text on the upper right side.



The above form has an unmapped component. To map it click “Click to map” text. Sisulizer shows Map Component dialog that lets you to choose the type of the component. If you component is panel choose *Basic panel* from the panel list.



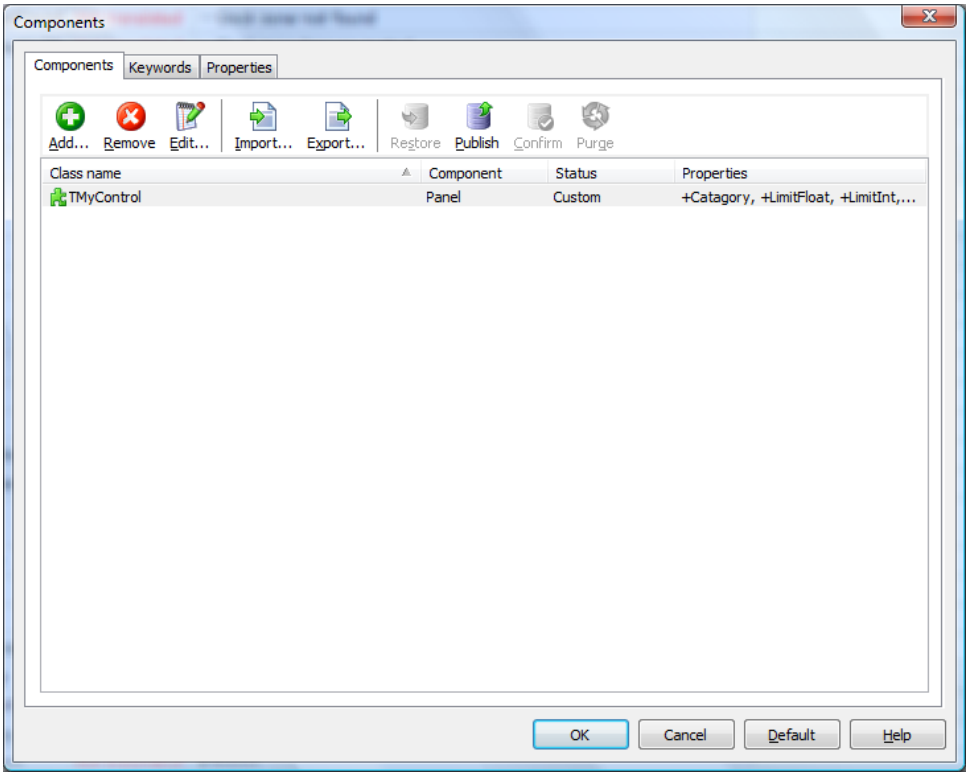
Sisulizer contains dozens of build in components. They cover most of the component types that applications use. However there are 3rd party components that are very complex and unique. Sisulizer does not have a counterpart for these components. In that case you better map the component to Basic control or Basic panel. In that case Sisulizer show the control as panel. The visual presentation might not be the right but keep in mind that Sisulizer can read and localize all properties of the component. Only the visual presentation of the control is limited. If the component is not visual component but a helper component such as TTimer map the component to Hidden component. In that case Sisulizer does not show the component at all.

When Sisulizer read form data it reads through all properties of all components. However Sisulizer ignores some property types that not needed for localization. By default it read layout properties such Left, Top, Width, Height and Align, font, IME, reading order, color and all string typed properties. This default property set is suitable for most components. Some components contain string properties that should not be localized and therefore should not be brought into project. For example SQL property in TTable component contains SQL statement that should not be localized in most cases. Component mapping are used to control what properties are scanned. You can exclude scanning of a property, include scanning of property that is not by default scanned and configure how property is scanned. Each component mapping contains zero ore more property mappings that can be used to control property scanning. Let’s look this through a sample. <sisulizer-data-dir>\VCL\Delphi\Mapping sub directory of Sisulizer directory contains a

sample that shows how to map and configure mapping of a custom property. The sample uses TMyControl component.

```
type
  TMyCategory = (mcNormal, mcHighPriority, mcFast);

  TMyControl = class(TPanel)
…
  published
    property Category: TMyCategory read FCategory write FCategory;
    property Description: String read FDescription write SetDescription;
    property LimitInt: Integer read FLimitInt write FLimitInt;
    property LimitFloat: Double read FLimitFloat write FLimitFloat;
    property Syntax: String read FSyntax write FSyntax;
  end;
```

Create a new project for Project1.exe. Click the “Click to map” label on visual editor to map the component to Panel. After you click OK on Map Component dialog Sisulizer prompts to rescan the project. Click Yes. This makes Sisulizer to rescan the project using the new mapping. TMyControl contains Syntax property that should not be localized. Right click Project1.exe on the project tree and choose Components. The Components dialog that is shown contains TMyControl. Double click it to edit mapping. Select Properties sheet to edit properties. Click Add button, type Syntax to Name edit check Excluded in the Mode group. This will instruct Sisulizer to ignore Syntax property of TMyLabel. Remember that by default Sisulizer reads all properties that are string typed. Out control contains also LimitInt and LimitFloat properties that are integer and float typed and not read by default. If we want to localize them we need to add them to the mapping. Click Add, type LimitInt, select Include in Mode group and select Integer number in Type combo. Similary add LimitFloat but choose Floating point number in Type combo. Finally add Category and set its type to Enumerated value. Now you have completed mapping of TMyControl component.



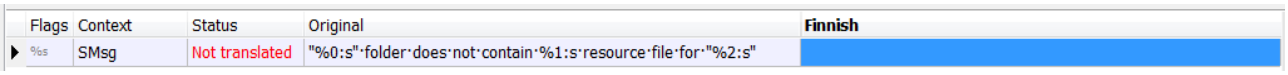
The mapping you have done apply only for this project. If you have other projects that use the same components you can publish the mapping to global mapping and they will be used for all other projects as well. Click Publish button on Component dialog to add mapping to global mappings (Tools | Platforms | VCL). If you use a 3rd party components that are not currently mapped map the component yourself. If you believe that your mapping should be included as default mappings of Sisulizer export your mappings to a file and send file to us. Click Export button to export mapping to an XML file.

Comments and other additional information

Sometimes it might be difficult to translate a resource string. What translator will see are the original string and the resource string name of the string. This may not give enough information to be able to correctly translate the string. Let's have an example. We have the following Delphi code:

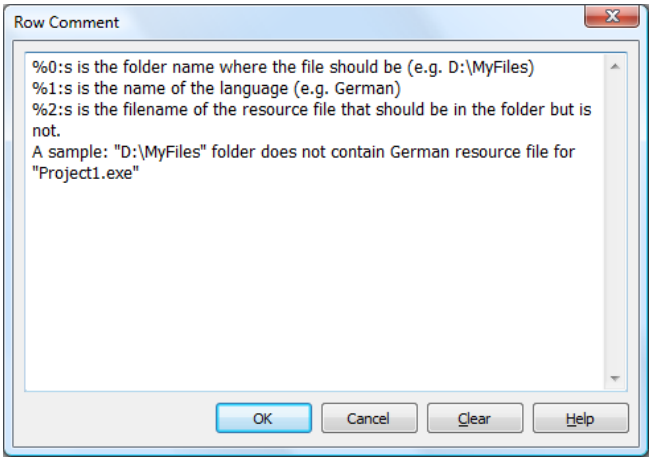
```
procedure WriteError(const folder, language, fileName: String);
resourcestring
  SMsg = '"%0:s" folder does not cointain %1:s resource file for "%2:s"';
begin
  ShowMessageFmt(SMsg, [folder, language, fileName]);
end;
```

Programmer will easily see that this should be something like this: "D:\MyFiles" folder does not contain German resource file for "Project1.exe". The folder parameter (%0:s) specifies the folder where the file should be (e.g. D:\MyFiles). The language parameters (%1:s) specifies the name of the language (e.g. German). The fileName parameter specifies the file where the resource file is meant for (e.g. Project.exe). Unfortunately translator can't see the above code. All he or she sees are the string and resource string name. The following picture contains the view the translator sees.

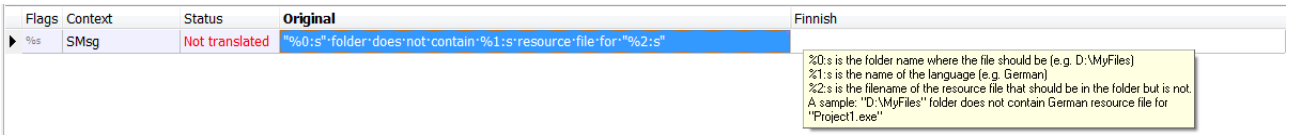


Flags	Context	Status	Original	Finnish
%s	SMsg	Not translated	"%0:s"·folder·does·not·contain·%1:s·resource·file·for·"%2:s"	

Programmer can add a comment for a row. This comment can contain instructions how to translate the string. To add a comment to a row right click the arrow on the left side of the row and choose *Row | Comment* menu. A comment dialog appears. Type the comment text.



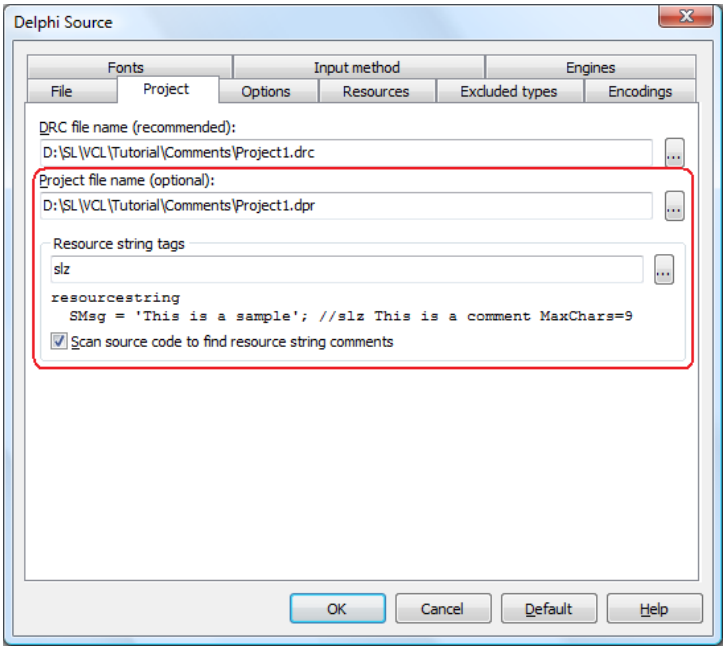
Click OK to close the dialog. Now a purple triangle appears on the upper right side of the original cell. It shows the user that there is a row comment. If you move mouse on the triangle Sisulizer will show the comment text on a yellow pop up window.



This works very well and it makes it possible to give instructions for translators. However adding comments manually might be difficult. Very often some other person but the developer create and maintain the project. He or she most like does not know the full meaning of the string. The developer that wrote the code knows the meaning perfectly. How can the developer add a comment? Fortunately Sisulizer has a feature that allows this. Developer can tag the resource string in the source code. Tagging means that developers adds a special comment to the same line to resource string. By default it is slz.

```
procedure WriteError(const folder, language, fileName: String);
resourcestring
  SMsg = '"%0:s" folder does not contain %1:s resource file for "%2:s"'; {slz %0:s is the folder
name where the file should be (e.g. D:\MyFiles)
%1:s is the name of the language (e.g. German)
%2:s is the filename of the resource file that should be in the folder but is not.
A sample: "D:\MyFiles" folder does not contain German resource file for "Project1.exe" }
begin
  ShowMessageFmt(SMsg, [folder, language, fileName]);
end;
```

Now the same text that was previously added manually to the Sisulizer project is written to the source code along with the actually resource string. The developer can easily add and maintain it. Most likely it will be updated if the developer changes the resource string. When Delphi compiles code it does not store comments anywhere so the compiled EXE file does not contain the comment. Sisulizer has to read it directly from the source code. In order to do that make sure that your Delphi source contains the Delphi project file. Right click your Delphi application file on the project three and choose Properties.



Project file name edit contains the project file. Make sure that you check *Scan source code to find resource string comments* check box. Otherwise Sisulizer does not scan the source code. Source code scanning shows down the overall Delphi source scanning speed so do not check the check box unless you really have resource strings comments.

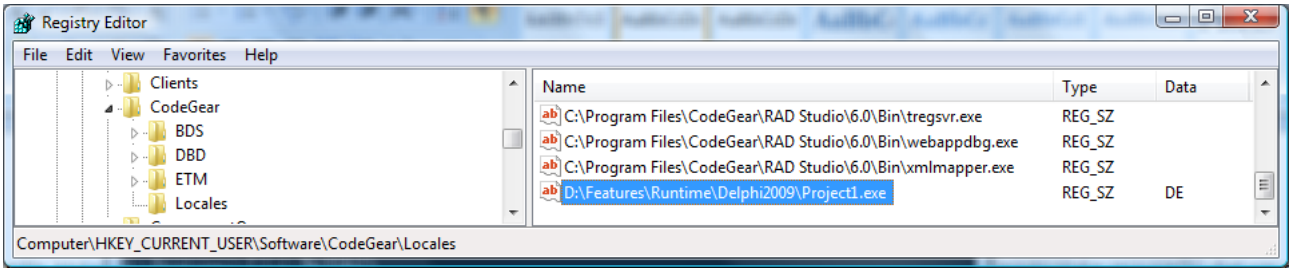
Resource DLLs

Until now we have used Sisulizer to create localized EXE. It means that Sisulizer has created one EXE file for each language. Each localized EXE is identical to the original EXE except the resource data is in different language. There is another way to localized VCL application: resource DLLs. When using resource DLLs you actually use two files: the original EXE and one or more resource DLLs. Only the EXE contains code (and original resources). Each resource DLL contains only resource data – no code. VCL's resource DLL files do not use .DLL extension but instead they use Windows language code or ISO language code (Delphi 2010 or later) as extension. If your original application is Project1.exe then Project1.DE or Project1.de is German resource DLL, Project1.DEU or Project.de-DE is German (Germany) resource DLL and Project1.EN is English resource DLL. The application loads a resource DLL that it wants to use and uses the resource data of the DLLs instead of EXE. Traditionally (using Visual C++) this has requires extra coding but with VCL no extra code is needed. VCL contains build in code that loads a resource DLL (if it exists). How does VCL choose what resource DLL to load if there are several resource DLLs in multiple languages. The following paragraph describes it.

First VCL looks if system registry contains information what resource DLL to load. VCL looks from following registry keys in this order:

1. HKEY_CURRENT_USER\Software\Embarcadero\Locales (Delphi XE and later)
2. HKEY_LOCAL_MACHINE\Software\Embarcadero\Locales (Delphi XE and later)
3. HKEY_CURRENT_USER\Software\CodeGear\Locales (Delphi 2009 and later)
4. HKEY_LOCAL_MACHINE\Software\CodeGear\Locales (Delphi 2009 and later)
5. HKEY_CURRENT_USER\Software\Borland\Locales
6. HKEY_CURRENT_USER\Software\Borland\Delphi\Locales

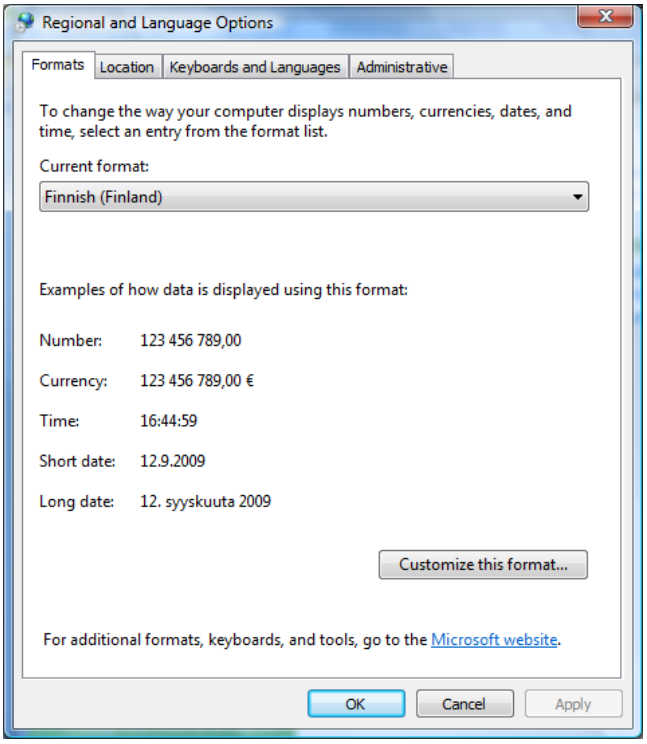
If a key contains the full path name of the EXE and language code then VCL will load a resource DLL having the same language code. For example if our application is D:\Features\Runtime\Delphi2009\Project1.exe and you want it to start in German add the following item to HKEY_CURRENT_USER\Software\CodeGear\Locales



The value of the key contains the full path name of the EXE and data value contains the resource DLL extension. In this case it is DE.

If there is not value in the registry or the resource DLL that was given there does not exist the VCL load the default resource DLL. This depends of your VCL version. Delphi up to 2009 tries to load a DLL matching to the system locale. This means if you have system locale German (Germany) VCL will search for de-DE or DEU. If the exact match does not exist then VCL tries to load country neutral resource DLL (e.g. de or DE instead of de-DE or DEU). Otherwise VCL does not load DLL but uses the resources of EXE. You can find the

above logic from VCL's source code. You can change default locale from Control Panel's Regional and Language Settings.



Delphi 2010 uses different approach. It tries to load the resource DLL matching the users default UI language. One way to change this value is to install multilingual user interface and the language you want to use. Another way is to add the locale override in the system registry as described above. If you use Delphi 2010 or later and you want to have the same behavior as Delphi 2009 (or earlier) you should add <sisulizer-data-dir>\VCL\LaDefaultLocale.pas into you project. Make sure the unit is added as first unit in the project. See <sisulizer-data-dir>\VCL\Delphi\Converter for sample how to use the default locale.

```
1  program Converter;

   uses
     LaDefaultLocale,
     Forms,
     Main in 'Main.pas' {MainForm};

   {$R *.RES}

10 begin
     Application.Initialize;
     Application.CreateForm(TMainForm, MainForm);
     Application.Run;
   end.
```

Look for <delphi-dir>\Source\Win32\rtl\sys\System.pas and look for LoadResourceModule functions. It performs the resource DLL loading.

Sisulizer contains LaResource.SetCurrentDefaultLocaleReg procedure that writes the currently active locale of the application that you are currently running to the system registry. Use SetDefaultLocaleReg to write locale of any application to the registry.

When your application starts VCL loads the default resource DLL according to the above rules. However the locale variables (e.g. CurrencyString, DecimalSeparator, ShortDateFormat, etc) are initialized based on the system locale of the computer. This is normally fine if the system registry does not contain any locale value for the application. If there is a locale value in the registry and it does not match the system locale of the computer the loaded resource DLL and locale variables are based on different locales. For example the user interface might be in German but date and times might be formatted based on UK standards if you run the application in German on UK computer. To make locale variables always matching the loaded resource add the following code into your main form.

```
initialization
  CheckLocaleVariables;
end.
```

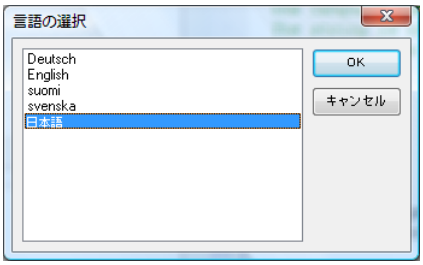
Whenever Sisulizer changes the language on runtime it will automatically update the locale variables unless you specify roNoLocaleVariables in the resourceOptions parameter of SelectResourceLocale or SetNewResourceFile. <sisulizer-data-dir>\VCL\Delphi\Formats sub directory of Sisulizer directory contains a sample that shows how to use CheckLocaleVariables procedure.

Runtime language switch (VCL only)

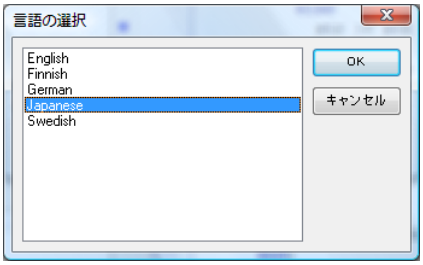
One useful features of resource DLL are that you can perform a runtime language switch when using them. In this case application will load the initial resource DLL on startup. Later user can choose another language and your application load a new resource DLL and translates its user interface. Standard VCL does not contain such a code but Sisulizer does. In order to support runtime change you have to add this code to your application. However the code is very minimal. In most cases you only have to call one function from your main form.

```
uses
  LaDialog;
…
procedure TMainForm.LanguageMenuClick(Sender: TObject);
begin
  SelectResourceLocale('EN');
end;
```

The above example contains event for Language menu. The event calls LaDialog's SelectResourceLocale functions that shows available resource DLL language and lets the user to select a new one. It requires one parameter that tells the original language of your application. Even your application only uses one function the actual Sisulizer code that performs runtime language switch is thousands of line in several units. Sisulizer's VCL directory contains these units and full source code. The above sample shows available languages in their own languages.



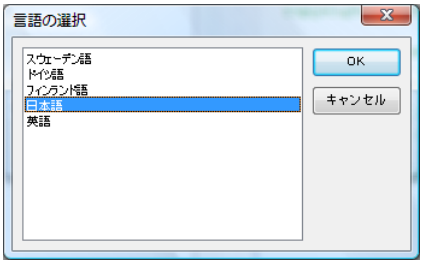
There are two other choices. The first is to show languages in language of your Windows. Second is to show languages in the currently active languages. Let's have a sample. We run an application in Japanese. If we call SelectResourceLocale with the above parameters we get a dialog like in the picture above. If we add [doUseCurrentLanguage] as the second parameter we get languages in English (picture below) if we run the application on an English Windows.



If we ran application on German Windows we would get names in German. If we want to show language names in the current language of the application (Japanese in this case) we need to work little bit. Write a RC file that contains string table. Add all your target language names there and give each name id that matched the language id. You can find language ids from Delphi's Windows unit. Search LANG_xxxx constants from Windows.pas file.

```
STRINGTABLE
BEGIN
  7  "German";
  9  "English";
  11 "Finnish";
  17 "Japanese";
  29 "Swedish";
END
```

Finally add the RC file into your Delphi project and compile. Now the language names are in the string resources and your translator can translate them.



<sisulizer-data-dir>\VCL\Delphi\RuntimeChange and LangChange samples show how to use localized language names.

If you want Sisulizer to save the active language to the system registry pass roSaveLocale in resourceOptions parameter. This way VCL remembers the active language of your application and next time you start it the application will start in the same language as in the previous time.

```
procedure TMainForm.LanguageMenuClick(Sender: TObject);
begin
  if SelectResourceLocale('EN', [], [roSaveLocale]) then
    Initialize;
end;
```

Let's look at what happens when a new language is loaded. SelectResourceLocale shows a dialog that list all the possible language (e.g. those that have corresponding resource DLLs). After use has selected the new language the functions calls LaResource.SetNewResourceFile function that loads the new resource DLL. After SetNewResourceFile has loaded the new resource DLL it translates existing forms because they are still in the language that was previously loaded. LaTranslator unit contains code to translate forms. When Sisulizer translates existing forms it will overwrite the current values with the values of localized DFM file. This means that after language switch the property values of your forms are back to their initial values. If you have changed the property values on run time you have to perform the changes again. A good practice is to write Initialize function in the form and call it from OnCreate event and after language switch.

```
procedure TMainForm.Initialize;
resourcestring
  SMsg = 'Sample Form';
begin
  Label1.Caption := SMsg;
end;

procedure TMainForm.FormCreate(Sender: TObject);
begin
  Initialize;
end;

procedure TMainForm.LanguageMenuClick(Sender: TObject);
begin
  if SelectResourceLocale('EN') then
    Initialize;
end;
```

Initialize sets the Label1.Caption property on run time so its value is not the same as in DFM. This is why the same change must be done after language change. Any form that you create after language switch will automatically use the new resource DLL and no additional steps are required. See <sisulizer-data-dir>\VCL\Delphi\Converter and LangChange samples to see how to implement runtime language change.

When TLaTranslator translates the forms it assigns the all property values of all components of all forms. This means the all property values are set to the translated value of the default values. Default values are those that you used on Delphi IDE when created the form. Most applications change these default values on run time as the above sample shows. If your application changes form sizes or positions on run time switching language might cause some flickering because first sizes and positions are set back to the default values and then Initialize function sets them to the final ones. You can prevent this disabling translation of some properties. The easiest way is to disable translation of all properties but string properties. To do that set LaTranslator.LaEnabledProperties to contain string properties.

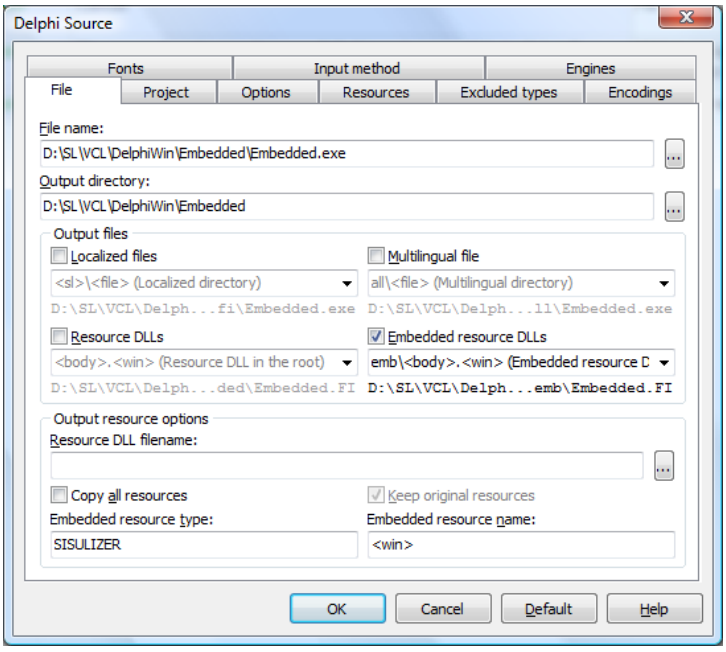
```
initialization
  LaEnabledProperties := STRING_TYPES_C;
end.
```

By default LaEnabledProperties is empty that means that TLaTranslator should translate all properties. If LaEnabledProperties contains one or more values, then TLaTranslator translates only properties whose type matches the types in LaEnabledProperties. This way to control translations if simple and cost effective but sometimes your user interface requires layout changes when translating it. This means that translator must have moved some of the components to make room for localization. If you then disable translation of Integer properties these translated positions will be ignored. In that case you better leave LaEnabledProperties empty and use LaBeforeTranslate event. if you assign a value to it TLaTranslator calls it before every time it tries to translate a property. The event passes information about property being translated. if you want to disable its translation set cancel property to True. If you want to change the translation value change the newValue parameter. The following code shows a simple LaBeforeTranslate event.

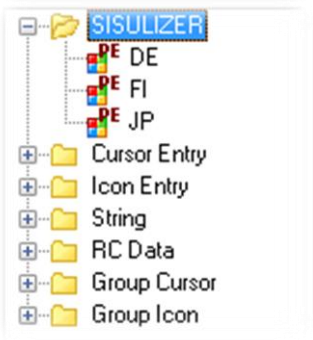
```
procedure BeforeTranslate(
  host: TComponent;
  obj: TObject;
  propertyInfo: PPropInfo;
  const currentValue: Variant;
  var newValue: Variant;
  var cancel: Boolean);
begin
  cancel := obj = Form1.Label2;

  if (obj = Form1.Label3) and (propertyInfo.Name = 'Caption') then
    newValue := UpperCase(newValue);
end;
```

The above code disables translations of Label2 on Form1 form. The event also changes the translation of the Caption of Label3 on Form1 form to upper cased string. See <sisulizer-data-dir>\VCL\Delphi\Events sample.

Using resource DLLs to implement runtime language switch is very convenient because it is the localization method CodeGear recommends. However the method is one drawback. It is that you no longer deploy only one file but you have to deploy the main EXE and one or more resource DLLs. Many developers want to make single EXE that requires no DLLs or other files. Fortunately with Sisulizer it is possible to do this. The solution is to use embedded resource DLLs. In this method the EXE file contains the resource DLL file inside the EXE in custom resource items. When EXE is run first time the application extracts the resource DLLs from resource items to actual resource DLL files. You get all the benefits of resource DLLs plus ability to deploy only one file. To make Sisulizer to embedded resource files right click your EXE on the project tree and choose Properties. In File sheet check *Embedded resource DLLs* check box. Next time you build your project Sisulizer creates emb sub directory and copies the EXE there and creates the resource DLLs for it and finally embeds them to the exe.



<sisulizer-data-dir>\VCL\Delphi\Embedded contains a sample that uses embedded resource DLLs. If you view the created EXE (emb\Embedded.exe) on resource viewer you will see that it contains SISULIZER resource(s): one for each language. These resources actually contain the resource DLL data. For example DE resource contains the data of Embedded.DE resource DLL file. The following image shows an EXE that contains German, Finnish and Japanese embedded resource DLLs.



You can also manually embed resource DLLs into your EXE. In that case make Sisulizer to create normal resource DLLs and then use SlAddRes.exe command line tool to add resource DLLs into your EXE as custom resources. SlAddRes tool is in Sisulizer directory.

The above instructions make Sisulizer to create EXE with embedded resource DLLs. You have to add one line of code to make your application to extract the embedded resource DLLs. Add the following lines of code into the initialize section of your main form.

```
initialization
  ExtractResourceFiles;
end.
```

LaResource.ExtractResourceFiles function extracts the resource DLLs from EXE to actually files if they do not exist or the existing files are older than EXE.

TLaTranslator can translate all basic properties such as string, integer, color, etc. However it cannot translate binary properties and some complex properties of certain components. In order to translate also those properties TLaTranslator uses modules. They are add-ons to TLaTranslator and provide translation of binary and complex properties. There are several module classes, one for each type of component. See <sisulizer-data-dir>\VCL\Delphi\Modules sample to see how to use modules. You can also write your own modules. See the source code of exiting modules to learn how to write modules. You can find all Sisulizer VCL source code from <sisulizer-data-dir>\VCL.

Online help, databases, data files and web pages

In addition of your application itself you normally have to localize other files in order to fully localize your product. Most applications have online help file. Sisulizer can localize HTML Help files (.chm). Just add your CHM file to the same project as you Delphi application. If you have web pages for your application and you want to localize them too you can still use Sisulizer. Sisulizer can also localize data of your application no matter if it is located on XML files, INI files or database tables. Sisulizer can be used to localize almost everything you need to be localized. Read Sisulizer's online help to find information about how to localize online help, web pages, data files and databases.

About Delphi and Sisulizer

We at Sisulizer know Delphi. Sisulizer itself is written in Delphi. Even Sisulizer can localize dozens of file formats from Delphi applications to .NET applications, from XML files or to Access databases, Delphi is always our prime target. For us Delphi is a first class citizen.