LangMan 2.6

The package of localization components for Delphi

Personal, Professional, Premium and Ultimate Editions

MANUAL

REGULACE.ORG

HW & SW Development

Table of Contents

1 General Information

LangMan components are used for very simple development of multilingual applications in **Delphi**. These components will literally relieve you of all troubles with programming of translations and switching between languages, and they maintain full clarity in the source code. In contrast to other solutions offered by the competition, **LangMan** is an absolutely unique and invaluable aid. **All available platforms** and compilers from **Delphi 2007** to current **Delphi 10 Seattle** are supported. **LangMan** can be used to develop **mobile, desktop** and **web** applications built on the **FireMonkey**, **VCL** or **uniGUI** frameworks.

You can develop your application in any basic language regardless of the necessity of future translation into other languages. Only in the case of strings used in the run-time of a programme it is necessary to refer to lexicons which form a part of this component package. In applications intended exclusively for Windows, **LangMan** is able to localize **resourcestrings** in the **Ultimate** edition during the run-time of the application. When designing forms, attention should be also paid to the variable length of strings. Everything else is fully automatic, therefore **LangMan** can save a significant amount of time in the development of multilingual applications.

LangMan components can be very easily incorporated into a finished project or a project in progress.

Using the **T(Uni)ResourcesTranslator(X)**[1] component you can very simply localize any **resourcestrings**, and thanks to the function of **import**[2] **of language data** you can also transfer a project localized by means of other tools into a project localized by means of the **LangMan** components in a very short time. For example, without having to make any adjustments it is possible to directly load data stored by the **TLang** component from the **FireMonkey** library.

The key component **T(Uni)LangManEngine(X)** is equipped with its own language editor which can be launched both in the target application and in a design in **Delphi**[3].

Although language files can be distributed simultaneously with the application, they can be distributed primarily independently, which makes it possible to modify or add a new language in the target application of the user. Another advantage of **LangMan** is its ability to inherit languages without any restriction on depth. One language can be inherited from another language, and that language can be again inherited from yet another language, and so on. It has the advantage that if a certain component has not been translated, the American version, for example, can proceed from the British one or vice versa, or the Austrian version from the German one, etc. At the same time this makes it possible to translate some languages only partially, which again saves a lot of time.

1 Only in the **LangMan Ultimate** edition.
2 Functions for importing language data are available only in the **Premium** and **Ultimate** editions.
3 The design-time language editor is available only in the **Premium** and **Ultimate** editions.

LangMan automatically creates functional links in the chosen menu for language selection. Alternatively, the user language selection can be included in the application by means of visual components **TLangCombo(X), TLangFlagSelectorX**[4] **(FireMonkey)** or possibly **TLangFlagsCombo**, or **TUniLangCombo**[5] in web applications. All of this does not require any work or complex setting. The individual languages can be assigned a graphic symbol or a flag which will be automatically displayed in the language selection menu or in the **T(Uni)LangFlag(X)** component.

1.1 Why you should use the LangMan components

- Absolutely simple use
- Implementation in an application will take minimum time
- Low procurement and maintenance costs
- Excellent technical support and fast troubleshooting
- **All source codes of a given edition** are included in all editions
- Support of platforms **Windows 32-bit, 64-bit, OSX, iOS, Android**
- After purchasing a new licence or an upgrade you have **all future versions of the relevant edition automatically for free at least for the duration of three years.**
- Thanks to the **run-time language editor**, users of your application can also participate in the development of localizations.

1.2 Supported Delphi versions

The **LangMan** components can be installed and used in the following versions of **Delphi (RAD Studio)**:

- 2007 (only ANSI coding of strings and without FMX components)
- 2009, 2010, XE (without FMX components)
- XE2, XE3 (without automatic **LanguageMenu** with FMX engine)
- XE4, XE5, XE6, XE7, XE8, X Seattle

We recommend that prior to purchasing the licence you first download free **LangMan Personal**, install it and try it out.

1.3 Installation of the LangMan components

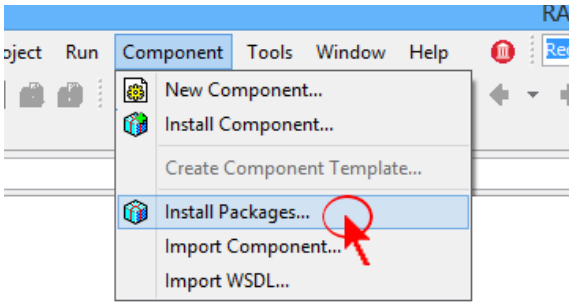
1.3.1 Before you begin

1) If you are installing the **LangMan** components on your current computer for the first time, you have to first create/select a folder anywhere on the disk where you will unpack the contents of the supplied ZIP archive. It is recommended that you create this folder next to the folders of other

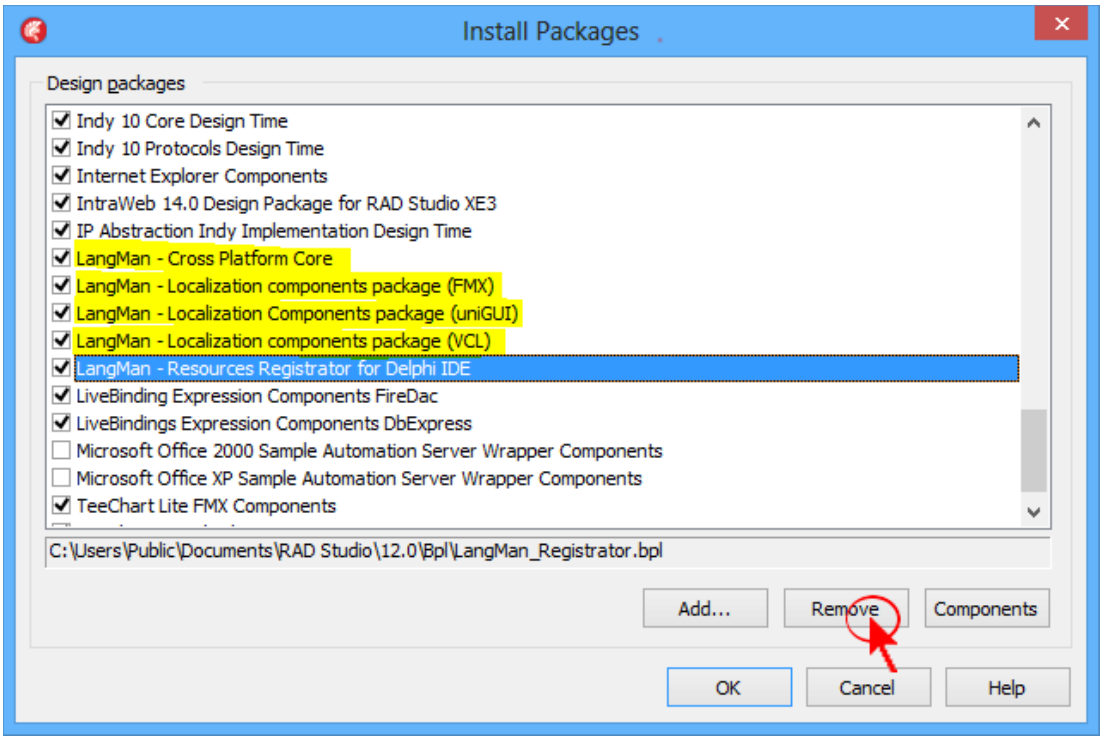
4 **TLanFlagSelectorX** is included only in the **Premium** and **Ultimate** editions.
5 **TUniLangCombo** is included only in the **Premium** and **Ultimate** editions.

components which you installed in the past or simply in a location which you will easily remember and where the files will not interfere with your work, because you will have to keep them there from the moment of their installation for the entire time of their usage.

2) If you already have any previous version of the **LangMan** components installed in **Delphi**, I recommend that you first remove all packages from the **Install Packages** menu.



This applies especially when upgrading to a higher edition and when changing the main version number, e.g. in switching from version 1.x to 2.x, etc.



You can unpack the ZIP archive with the new version of components in the same folder where the previous version was. You will simply rewrite the original files.

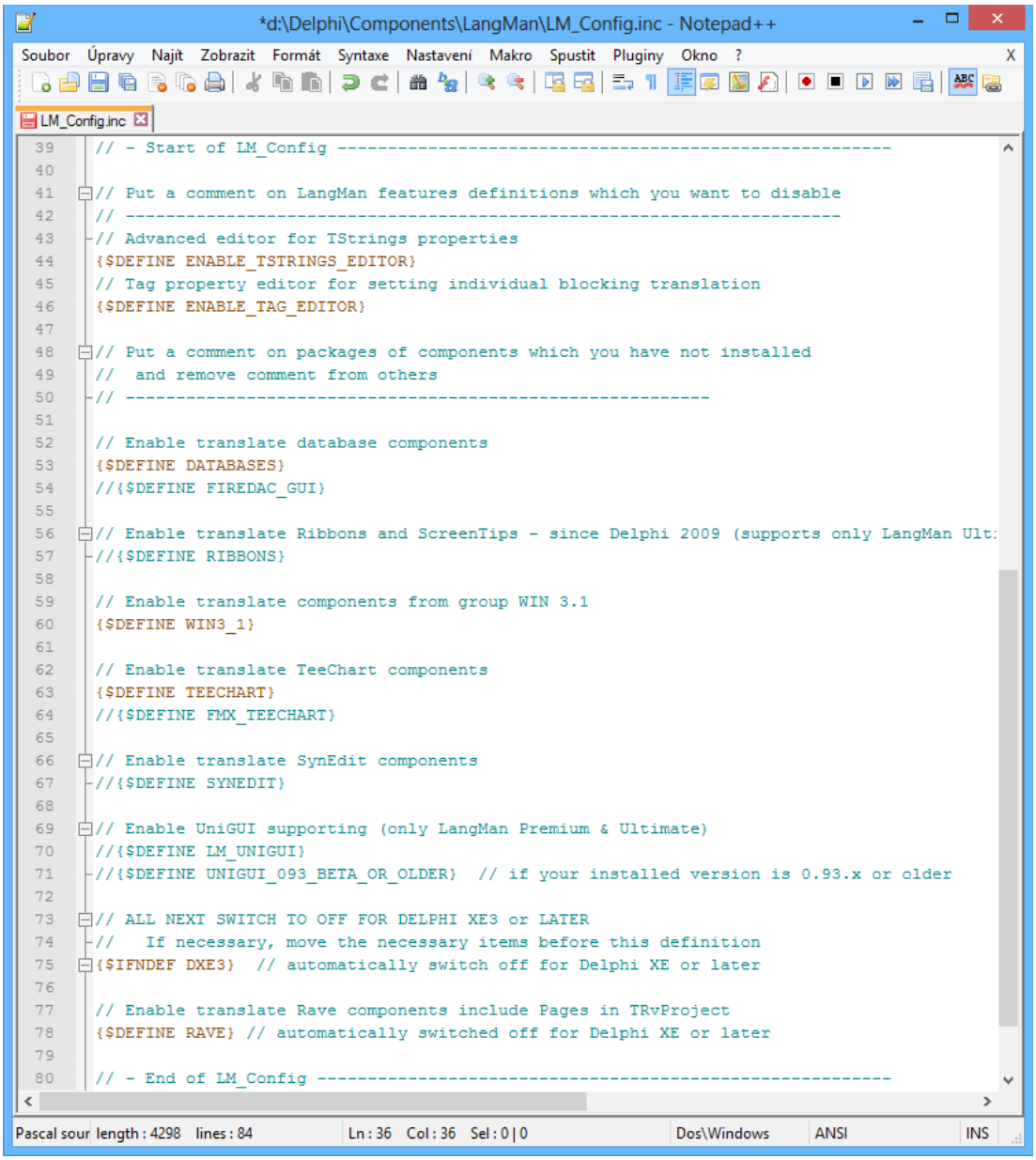
1.3.2 Preparation for compilation

After you have read the licence agreement and downloaded the ZIP archive of **LangMan** components, unpack the archive into the predetermined folder.

The following location has been chosen for this installation example:

`D:\Delphi\Components\LangMan`

Now you have to modify the **LM_Config.inc** file which you will find among other unpacked files. In the given file you will find several definitions which activate different **LangMan** functions/components. By commenting a certain line with a definition you will deactivate the relevant function, and vice versa.



It is particularly important to comment definitions permitting the translation of those component sets which you do not have installed in **Delphi**, because otherwise it would be impossible to compile the **LangMan** components.

For example, if you know that you do not have installed **TeeChart** components, comment the "TEECHART" definition, or on the contrary if you have them installed and you wish to have the option to localize, for example, **uniGUI** components, uncomment the "LM_UNIGUI" definition.

Do not forget to save your changes in the **LM_Config.inc** file.

Now start **Delphi (RAD Studio)**, or possibly close the currently open project and in the **File** menu click on **Open Project**.

Open the file according to the following criteria:

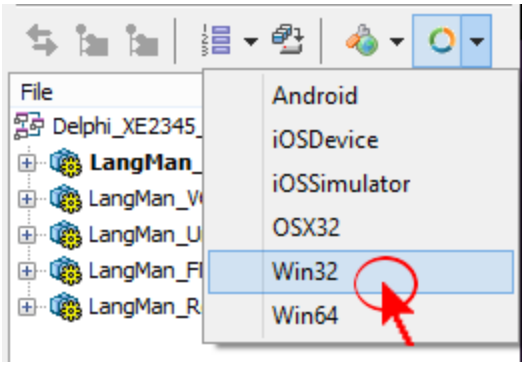
- `Delphi_XE2~10_LangMan.groupproj` **(in Delphi XE2 – X10)**
- `Delphi_2009~XE2_LangMan.groupproj` **(in Delphi 2009 – XE2)**
- `Delphi_2007up_LangMan_VCLonly.dpk` **(in Delphi 2007 – XE5)**
  - for the installation of LangMan components for VCL only
- `Delphi_2007up_LangMan_VCLuni.dpk` **(in Delphi 2007 – XE5)**
  - for the installation of LangMan components for VCL and uniGUI
  (only in the Premium and Ultimate editions)
- `Delphi_2007_LangMan_Ultimate.dpk` **(in Delphi 2007)**
  (LangMan Ultimate for Delphi 2007)

Confirm possible reports on the package conversion or removal of references to missing project files.

1.3.3 Compilation of source files

The following procedure differs only slightly depending on the **Delphi** version and the edition of **LangMan** components.

In **Delphi XE2 and later versions** you choose the target platform before compilation (owners of older versions do not have this option):

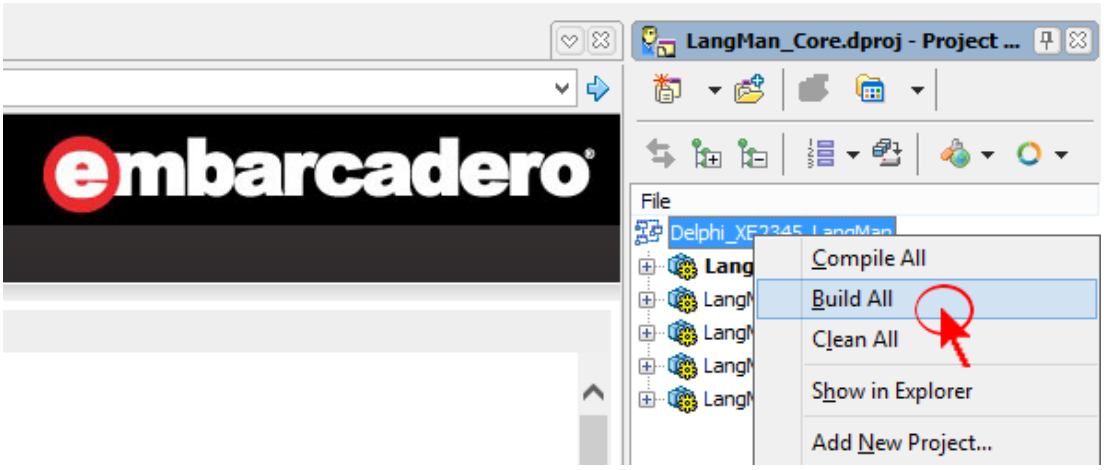


Start by choosing the **32-bit Windows (Win32)** platform. After the installation you can choose any other platform and compile the **LangMan** components for it.

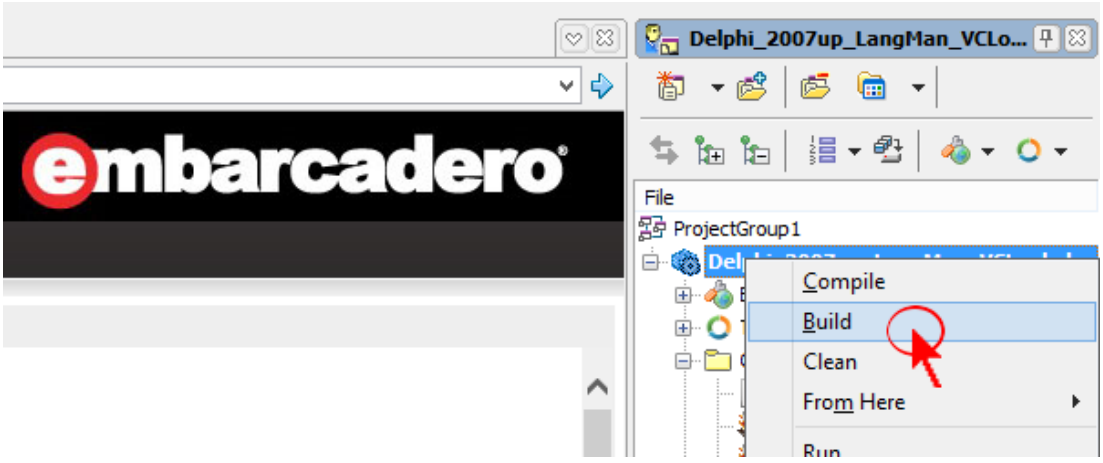
Note: *In order to make components accessible for the required platform, they have to be compiled and installed also for the* ***Win32*** *platform.*

Now compile the entire project group.

1) If you opened a file with the extension **.groupproj** in the previous step, use the right mouse button to click on the project group name in **Project Manager** and select **Build All**.



2) If you opened a file with the extension **.dpk** or **.dproj** in the previous step, use the right mouse button to click on the name of the open package in **Project Manager** and select **Build**.

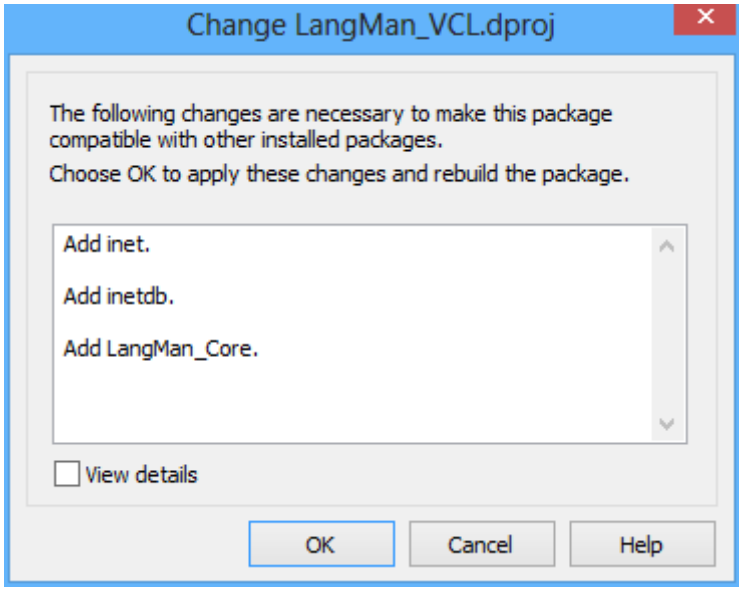


Compilation and building of libraries will follow.

No serious errors should occur in the course of the compilation. If any missing unit or library is reported, you may have forgotten to ban some function in the **LM_Config.inc** file, therefore try to return to this step. If the report seems unjustified, check if the relevant unit or library is compiled also for the selected platform.

However, if there is a report of a required import of missing libraries similar to the one in the below given figure, you probably do not strictly follow these instructions. It does not matter, you can either confirm the proposed changes and try to build the package again, and if this does not work, you can close

Delphi, unpack the once more downloaded ZIP archive in the same folder and repeat the procedure right from the beginning.



Similarly, if you find the following type of warning in the list of error reports:

```
W1033 Unit 'xxx' implicitly imported into package 'yyy'
```

uninstall all **LangMan** packages according to the description in chapter 1.3.1 provided they have been installed, unpack again the ZIP archive and repeat the procedure.

The given errors may occur, for example, if the package or project file has been changed. Do not therefore open settings from the **Project / Options...** menu and do not add or remove project files, etc.

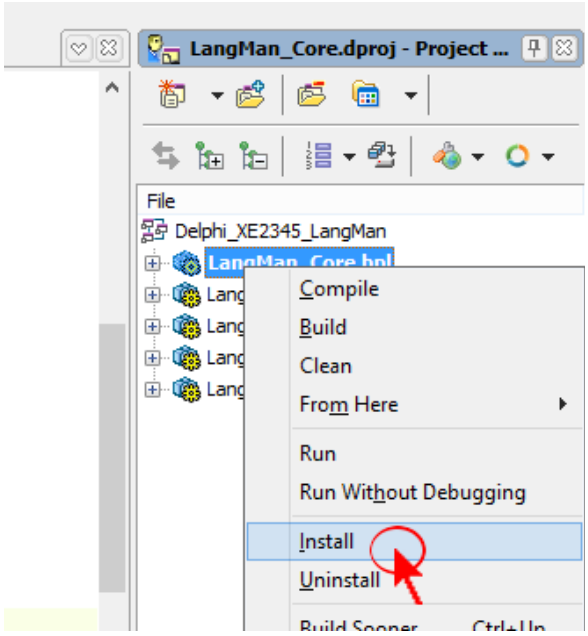
If you fail to translate **LangMan** even after repeated attempts, do not hesitate to contact the author of the components.

1.3.4 Installation

Libraries with the **.bpl** extension will be created after successful translation of source codes for the **32-bit Windows (Win32)** platform. Their location is not very important for you. You still have to install them for their use in **Delphi**.

In **Project Manager** click the right mouse button on the first package and select **Install**. If more packages are present in the group (the **.groupproj** file is open), you will install all packages one by one in the same way.

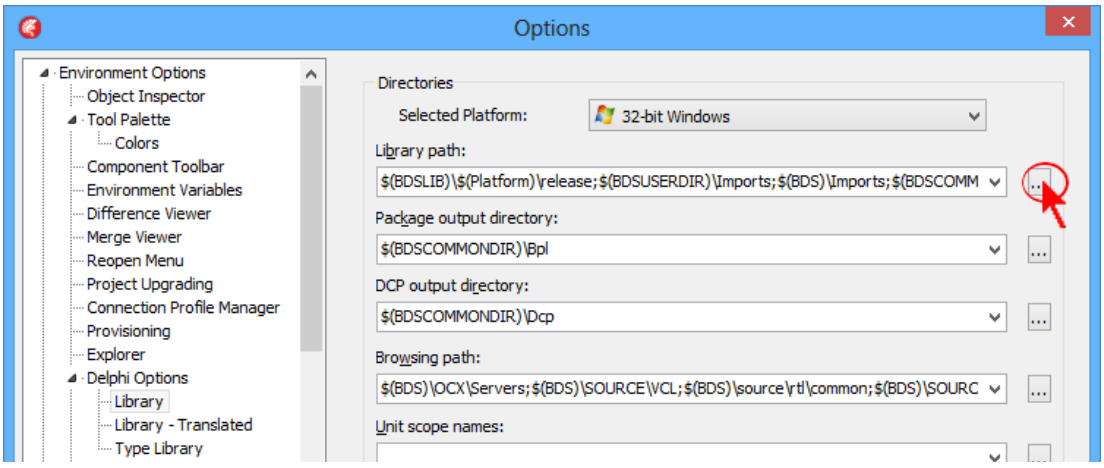
You do not perform this installation step for the other platforms. If the components are installed for the **Win32** platform, it is enough to compile the source codes for the other platforms. However, you need to take into account that in order to compile the **LangMan** components for the required platform, the libraries on which the **LangMan** components are dependent have to be also compiled for this platform.



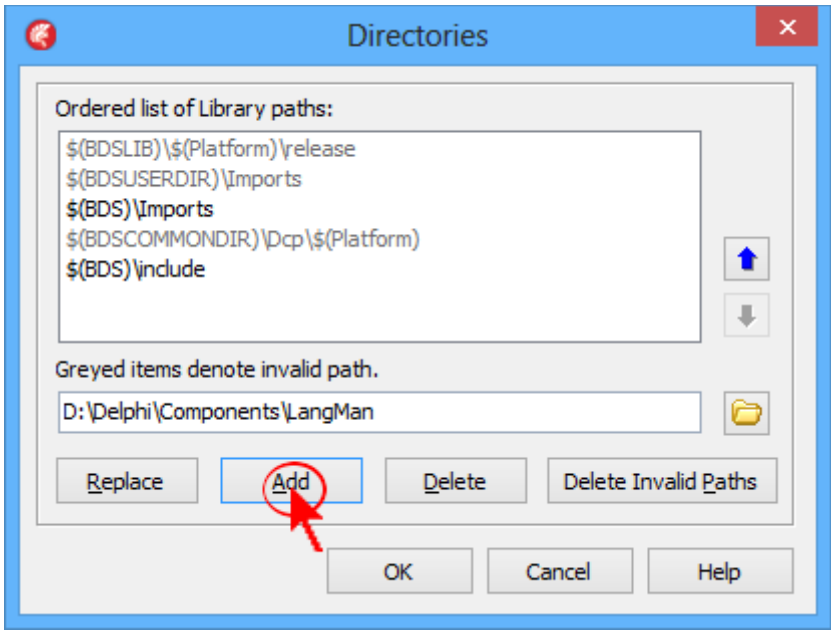
1.3.5 Adding a path to the LangMan files

In order to successfully finish the initial installation you still have to add a path to component files in the **Library path** setting.

Click on **Tools** / **Options...** in the main menu. In the **Options** window find the **Library** item, which is hidden under **Delphi Options** and click on the button marked by three dots behind the **Library path** field:



Insert or type the path to source files of the **LangMan** components in the edit box and click on the **Add** button.



Confirm and save the setting by the OK button. **The installation has been finished.**

2 LangMan 2.6 Package Components

The entire set of **LangMan** components can be divided into:

1) Nonvisual components used directly for localizations of applications into various world languages. They include:

T(Uni)LangManEngine(X), **T(Uni)LangManClient(X)**, **T(Uni)ResourcesTranslator(X)***, **T(Uni)DesignedLexicon(X)**, **T(Uni)ProgrammableLexicon(X)** and **T(Uni)InlineLexicon(X)***

2) Visual components for signalization of the selected language, language selection and possibly also for opening the language editor:

T(Uni)LangCombo(X), **TLangFlagSelectorX**, **TLangFlagsCombo** and **T(Uni)LangFlag(X)**

3) Bonus visual components which are useful for the development of multilingual applications:

T(Uni)ValuedLabel(X) and **TLangManRichEdit**

4) The last group also includes the class **TLangManStrings** which is not a component and which can be used for generating a localizable text in **TMemo,** etc., or in any object which is a descendant of the **TStrings** class.

Components which are compatible with **FireMonkey** are designated with the letter "**X**". Components which are intended for **uniGUI** web applications are designated with "**Uni**". You can find components with "**X**" and "**Uni**" only in the **Premium** and **Ultimate** editions.

The function and purpose of components with and without "**X**" or possibly "**Uni**" is the same. They differ only in the supported framework and in details associated with the relevant framework. In this document the text describing the individual components and their functions is, therefore, often common for all three components which differ only in their target framework.

2.1 *T(Uni)LangManEngine(X)*

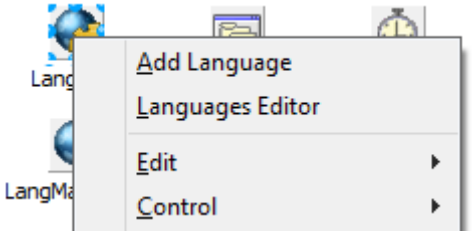
T(Uni)LangManEngine(X), further referred to as the engine, is the most important component of the **LangMan** package. It provides translation, switching between languages, it manages the language files and includes the language editor. This component is necessary for functioning of all the other language components, except the **T(Uni)ValuedLabel(X)** and the **TLangManRichEdit** component. In most cases, one language is common to all the parts of the application, so it is necessary to place the engine once into the unit (on the form or, rather into the **TDataModule**), which will be accessible from all the other forms. In case you need to separate two or more parts of the program considering the language, for example if you want to choose the program language and the printout language separately, you can use as many engines as necessary. Individual engines must differ in **LangFileSignature**, **LangFileExtension** and **LangSubdirectory** properties settings. A single difference in some of the given characteristics is sufficient

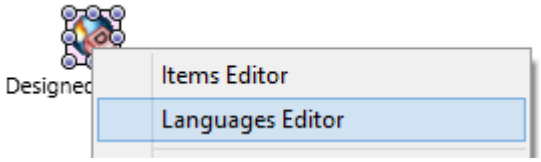
* **T(Uni)InlineLexicon(X)** and **T(Uni)ResourcesTranslator(X)** are included only in **Ultimate Edition**

to the distinction. These characteristics determine the basic parameters of language files that will belong to given language engine.

The built-in editor can be launched during the programme run-time so that even the end user, if allowed, may add or possibly adjust languages.

In the **Premium** and **Ultimate** editions, the editor can be used even during the design of an application in the IDE. It is made accessible by means of local menus of the components **T(Uni)LangManEngine(X)**, **T(Uni)LangManClient(X)**, **T(Uni)ResourcesTranslator(X)** and all lexicons.





The **T(Uni)LangManEngine(X)** component on its own is unable to read strings from the project which should be localized. This is performed by the following components.

2.2 *T(Uni)LangManClient(X)*

This component, further referred to as the client, serves for installing the form, on which it is located, into the translation by selected engine. In addition, it allows you to choose which properties on the form and its components are supposed to be translated. If it is desired for some properties to be translated by one engine, and the others by another engine, it is possible to place more clients on one form and to assign different engines to them.

2.2.1 What will be translated

The purpose of the **T(Uni)LangManClient(X)** component is to create a certain list of components and their textual properties and to hand it over to the engine. This list is created automatically at the moment the form is completed or by calling the **RecreateTransStruct** method. Further down in the text you can also read about the possibility of adding individual items additionally by means of the **AddComponent** method. The items of the mentioned list will be located in the language editor and it will be possible to localize them. The programmer has several options how he can influence what components and properties will be added in this list:

1) By setting the properties:

TransStringProp, **TransTStringsProp**, **TransStructuredProp**,

TransOtherProp, **TransAdditions** and **TransXtraOptions (in FMX)**,

by means of which you determine which properties can be translated in the scope of the entire form. The ***false*** value completely prevents the translation of the relevant property and the ***true*** value means that the relevant property, (e.g. Text "**(f)pnText**"), or possibly the entire component (e.g. TChart "**(f)cnTChart**"), can be translated if it fulfils the requirements given in the following points.

2) The property has to include characters with a numerical value higher than or equal to character "**A**". In other words, numbers and certain special characters are ignored.

3) The property has to include a string which differs from the **Name** property. This condition is not checked in certain exceptional cases, in particular in more complex data types.

4) In components with set **Action**, a property loaded from the relevant action has priority; therefore, for example, **Caption (Text)** from **TMenuItem** will not be inserted in the list if it replaces **Caption (Text)** from the assigned action.

5) The component has to be supported. If it is not supported, it is possible to simply add functionality for the translation of your own and non-standard components of third parties by means of the **LMVCLAdditions** unit and **LMFMXAdditions**. The procedure is described directly in the header of these units. Mostly it suffices, however, that the unknown component is created from some standard component, in that case the translation of all inherited properties from the supported component will work.

6) In components proceeding from the **TStyledControl** class, which applies to the majority of visual components of the **FireMonkey** library, the **AutoTranslate** property has to be set to the ***true*** value. Otherwise, the relevant component will not be translated.

7) The **VCL** components and some **FireMonkey** components do not have any property which would be similar to the above mentioned **AutoTranslate**; there is therefore also the option of restricting the translation of components by means of the **Tag** property. When you open the menu of the **Tag** property, it will offer you one or more options for individual blocking of the translation of the given component.

"Do not translate" (-799) – the component will not be translated

"Do not translate Text" (-797) – translation of the **Text** property is disabled

"Translate Text only" (-790) – only translation of the **Text** property is allowed

"Translate Hint only" (-789) – only translation of the **Hint** property is allowed

"Do not translate Hint" (-798) – translation of the **Hint** property is disabled

"> Reset <" – **Tag** is reset and it can be used for any purpose

2.3 *T(Uni)ResourcesTranslator(X)* ***(only in the Ultimate Edition)***

The component **T(Uni)ResourcesTranslator(X)** is used for localizations of

resourcestrings which have been linked to the application. In **Delphi** these constant strings can be defined in several ways. Most frequently by means of the keyword **resourcestring** directly in the source code of the **.pas** unit or by means of a block:

```
STRINGTABLE
BEGIN
     index, "string"
END
```

in the **.rc** file which you then add in your project.

In the case of strings used in the code which need to be localized it is better to use one of the lexicons supplied together with **LangMan**. However, the entire range of **resourcestrings**, e.g. for exceptions, various error reports and also for dialogue box buttons (**MessageDlg**, **ShowMessage**), is defined directly in the standard units of **Delphi**. Even some of the standard visual components, e.g. **TRibbon** for elements in the Menu or **TGesturePreview** for displaying the "**StartPoint(s)**" legend**,** use **resourcestrings**. Therefore if you wish to have really everything localized in your application or you use **resourcestrings** for any reason, you will need this component to localize these strings.

2.3.1 How it works

An experienced programmer knows that data stored in the resources (in the case of Windows applications directly in the exe file) are not very suitable for transcribing. **LangMan** therefore works in a slightly different way. After the start of the application it replaces routines used for accessing the strings in the resources by its own code and during every reading of a certain string from the resources it always returns a localized form according to the selected language. Unfortunately, this procedure has been written only for Windows applications as of now, that is, for target platforms **32-bit Windows** and **64-bit Windows**. Although the use of the **TResourcesTranslatorX** component in applications which are translated for other platforms will not cause any errors, **resourcestrings** will not be localized in these cases.

2.3.2 Selection of translated resourcestrings

After adding the **T(Uni)ResourcesTranslator(X)** component into your application you have to choose which strings should be localized (offered in the language editor for translation). There are thousands of them in **Delphi**, but no application is as complex as to use all of them, far from it in most cases, some strings are not used at all. Therefore you should choose only those which may be used in your application or only those which you wish to localize. At the same time it is good to know where each string is used in the code because in some cases a change during the run-time of the programme may have a negative effect on the proper function of the programme.

You will register those **resourcestrings** which you wish to localize by **ResourcesTranslator** by means of a specialized editor (registrator) which you will display by clicking on the **RegisteredStrings** property "**...**" button or

by double clicking on the component icon, or possibly from the popup menu of the component – the **Translated resources** item.

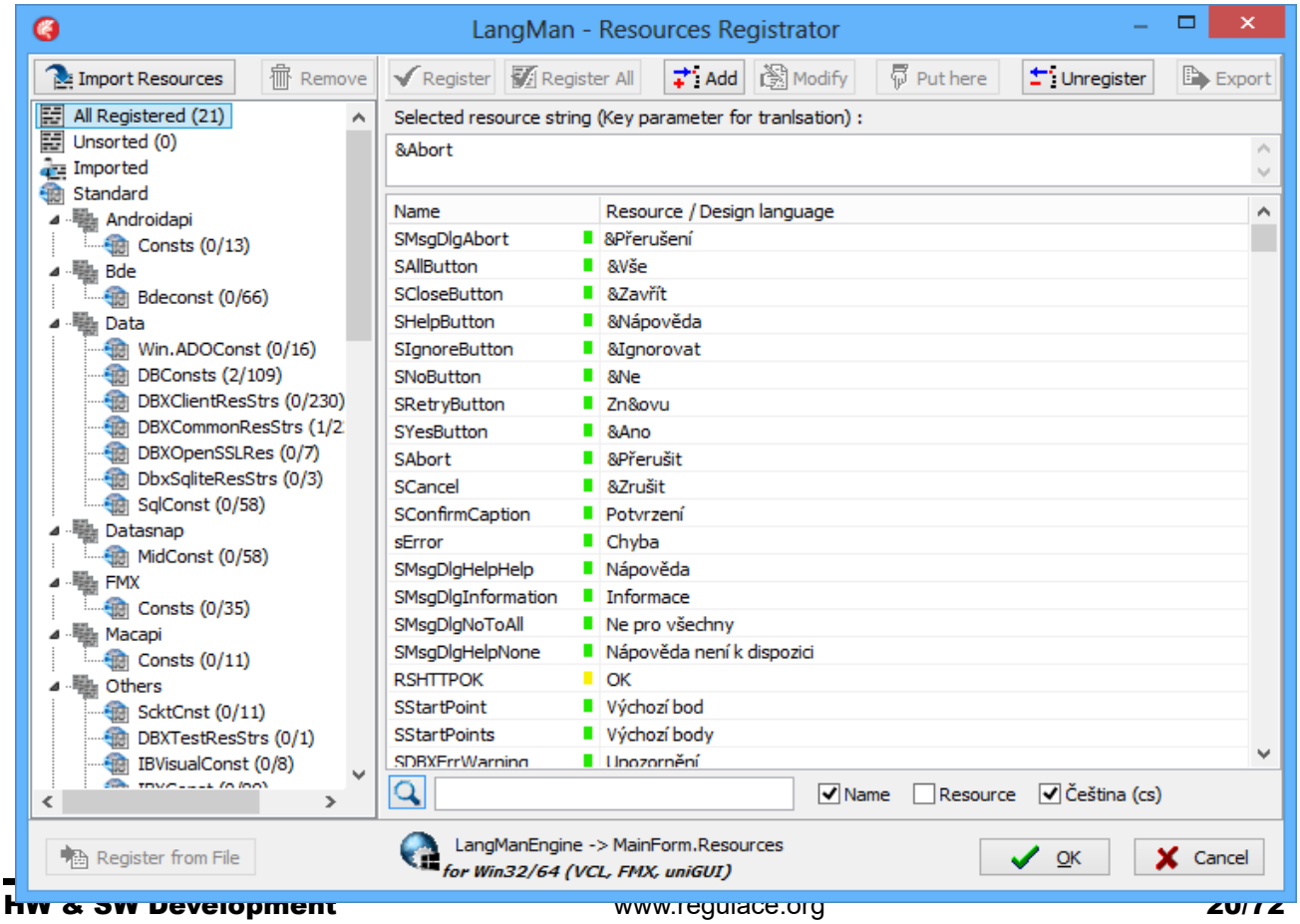
In the left part of the editor window there is a tree list of groups (units) from which **resourcestrings** can be registered for localizations.

All Registered - All registered strings are displayed.

Unsorted - Strings not sorted into any group. You can simply add individual **resourcestrings** here, for example, if you do not want to create an independent group because of one item. This group can be renamed.

Imported - Manually added groups (units) are inserted here by means of the **Import Resources** button. The editor will automatically load defined items from them. If you do not register any item in this unit, it will be automatically removed after the editor is closed.

Standard - A list of standard **Delphi** units which contain **resourcestrings** in a language set by the **OriginalLanguage** property. Some units have been intentionally excluded from this library because they are intended for the design and they do not manifest themselves in the application run-time. They include, in particular, units for **Databinding**, **DUnit**, **ToolsAPI** and **Property Editors**. The standard languages supported by **Delphi** are: **en** – English, **de** – German, **fr** – French and **ja** – Japanese. If a different language is set and no standard strings are registered, the **Standard** group will be empty.



You will register the required strings simply by selecting them and by clicking the **Register** and **Register All** button, by clicking on the name of the item or by transcribing the string.

As you do not always develop applications in the same language in which the already existing **resourcestrings** are defined, it is necessary to use this editor at the same time to carry out localization in the design language. This language is a language which you have selected for the development of the application and which you have set in the **DesignLanguageName** property of the relevant engine. If you are developing the application in English and the strings in the resources are also in English, then you only need to register them.

By means of the **Add** button you can manually add your own item in the selected group. The key parameter for localizations is always the original form of the string as it is defined in the programme. The names of items are used only for better orientation and to make search easier.

Manually added items not bound to a known string from an imported or standard unit can be modified by clicking on the item name or the **Modify** button.

The **Put here** button is used for transferring an item registered in another group into the currently selected group. These items are marked in blue.

In order to save work in the development of further applications, you can also register multiple strings by using the **Register from File** button. A file exported by means of the **Export** button from another **T(Uni)ResourcesTranslator(X)** or an ordinary language file, or language export, can be used for multiple registration. An obvious condition is that the source file for multiple registration should include localized items of registration in the same language as is the design language of the current project.

2.4 Lexicons

Except to the automatic translation of static components, located on the forms, strings for dynamically created dialog boxes, notifications, etc. are often needed in the programs. For this purpose lexicons are used where necessary strings may be defined. These strings are then automatically translated, so the particular item is always read in the appropriate language.

2.4.1 T(Uni)DesignedLexicon(X)

Strings in **T(Uni)DesignedLexicon(X)** are defined by using the editor directly in object inspector. Each strings have indexes that you use to get in the localized form of the string. Editor works like editor of **TStrings** class, where the first line starts with index 0 and goes 1, 2, etc., with the difference that when removal of any line (string) does not renumber the following records. At any later time may be empty index restore by pressing **Insert key** on the line editor.

2.4.2 T(Uni)ProgrammableLexicon(X)

Programmable lexicon items can be defined, unlike **DesignedLexicon**, in the program source code, for example in the **OnInitialization** event method. This lexicon is applied in cases where the language strings are known only during the program run-time. A condition for translating is that all items that are supposed to be translated automatically have to be in the lexicon before editing the language, otherwise it will not be possible to locate them into other languages. As a result the items of this lexicon will not be visible in the language editor at design-time.

2.4.3 T(Uni)InlineLexicon(X) (only in the Ultimate Edition)

You can define the items of this lexicon literally **inline** directly in the source code by means of the **Loc['**text**']** property or in the case of a link by means of the **SLink['**text**']** property. The **Inline lexicon** differs from the preceding ones in not using indexes but directly independent strings written in the source language of the design. You, however, do NOT have to initialize the items of this lexicon prior to the editing of the language (as the case is in **ProgrammableLexicon**) because they will implement themselves in the translator already during the compilation of the application. The undeniable advantage of this lexicon is of course the fact that you do not have rely on some indexes and remember which string you have under which index at all times, you only use a string directly in the source code and do not have to do anything else. The only condition is that you have to use a directly unnamed constant string designated by quotation marks in the square brackets of the **Loc** and **SLink** properties. It is not possible to transfer a variable or an indirectly defined constant. In such cases initialization of these entries would be necessary before language editing.

2.4.4 Other lexicons features

Another function of all lexicons is translating of dynamically generated extensive texts. Only links to lexicon strings instead of individual strings may be inserted into these texts during the dynamic creation of the content. This feature of the lexicon is called **Link** and **SLink**[*]. By means of **CompleteString** method it is possible to transfer the text with links into the currently selected language at any time. This feature is also used by **TLangManRichEdit** component.

2.5 *T(Uni)LangCombo(X)*

This is a standard **ComboBox**, which is after assigning to the engine automatically filled with existing languages. When selecting a language, all clients and lexicons will be translated. In case that it is allowed in language engine to create or edit languages, the selection is extended by these options (applies only to **TLangCombo** and **TUniLangCombo** in VCL mode).

* Supported only in **Ultimate Edition**

2.6 TLangFlagsCombo

TLangFlagsCombo is an enhanced **ComboBox** whose items are extended by language icons and possible options for creating and editing the languages by the user. Outside of these icons the function and appearance is identical with **TLangCombo**.

Note: *In **FireMonkey** the corresponding component may be easily created by combining the **TLangComboX** and **TLangFlagX** components.*

2.7 T(Uni)LangFlag(X)

This component is a descendant of **TImage (TUniImage)**. Can be linked to engine, which automatically sets the **Picture** property (or **Bitmap** and **MultiResBitmap** property on **TLangFlagX**) according to the currently selected language.

2.8 T(Uni)ValuedLabel(X)

T(Uni)ValuedLabel(X) is an additional component similar to the standard **TLabel (TUniLabel)**. It has **ValueName**, **ValueSeparator**, **ValueSpaces** and **Value** properties, instead of the **Caption (Text)** property. These properties are linked in the final display, while **ValueSpaces** indicates the number of spaces instead of **ValueSeparator**. The trick is that **LangMan** translates except **Hint** (VCL) only the **ValueName** property. It is sufficient to enter only the "**Value**" value in the program. The usage is then clear from this description.

2.9 TLangManRichEdit

TLangManRichEdit is a descendant of **TCustomRichEdit** and was developed for generating various statements, logs, reports, etc., with various font styles. It is similar to class **TLangManStrings** (see chapter 4.14 and 6 of this manual). Compared to **TLangManStrings** this component allows to create nicer and comprehensible documents, making it possible to dynamically change the language of contained text. An important difference compared to **TRichEdit** is the impossibility of edit the text by user. The property **ReadOnly** is hidden and set to ***true***. To write and edit text content are in **TLangManRichEdit** implemented the necessary methods. The advantage of this component is that you can preset a stales set and when writing only refers to this styles. Of course there is also possibility to save the resulting document as RTF.

2.10 TLangFlagSelectorX

TLangFlagSelectorX is a scrollbox with the flags buttons of all available languages. After clicking on the flag will be switched the application to the appropriate language. This component does not require any programming. Allows horizontal and vertical orientation with adjustable flag spacing. On **TLangFlagSelectorX** can be placed any effect component. This effect is automatically applied to the flag of the currently selected language.

3 Language editor

The **LangMan** component set includes a language editor which can be opened during the run-time of an application by means of **ShowLangEditor** and **ShowLangCreator** methods of the **T(Uni)LangManEngine(X)** component or directly from an automatically created language menu as a submenu of the **TMenuItem** object assigned to the **LanguageMenu** property, or from the menus of visual components **TLangCombo**,**TLangFlagsCombo**, **TLangFlagSelectorX** and **TUniLangCombo**, if visibility of editor items is allowed in the menus by means of the **LangCreatorVisible** and **LangEditorVisible** properties. The **TLangComboX** component (in FireMonkey) does not have this option and in the **uniGUI** web application the editor can be opened during the run-time only in the VCL mode.

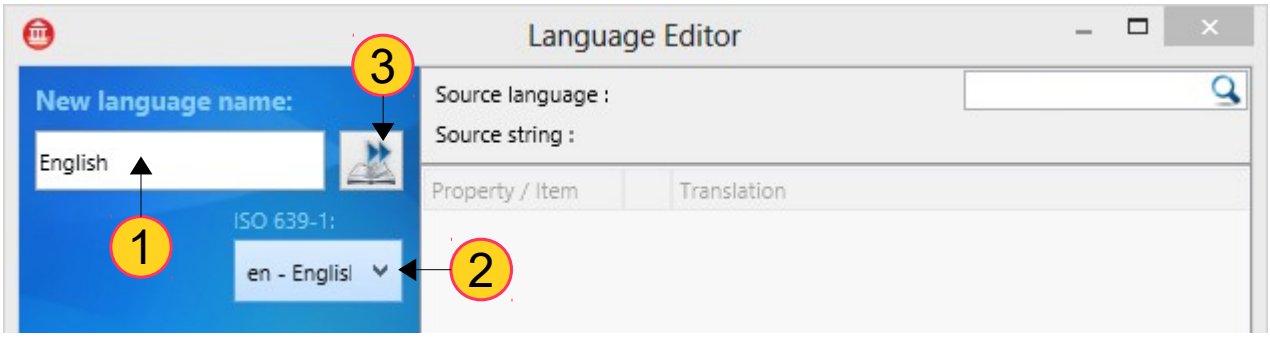
In the **Premium** and **Ultimate** editions the editor may be launched also in design-time directly in the Delphi IDE.

It is important to note that the language editor always includes only those items which are in the memory. Therefore, you will not find there forms which were not created in the memory.

3.1 Editor window

The editor has been designed in such a way that its use is maximally intuitive, therefore it is probably not necessary to describe it in detail. We will rather focus on some of the important features of the editor and on the description of properties which are not completely clear.

If you launch the editor by a command for the addition of a new language, you will first have to select a source language (if there are more than one in the application) and then enter the name of the new language **(1)** and a two-letter ISO code **(2)** pursuant to international standard ISO 639-1.

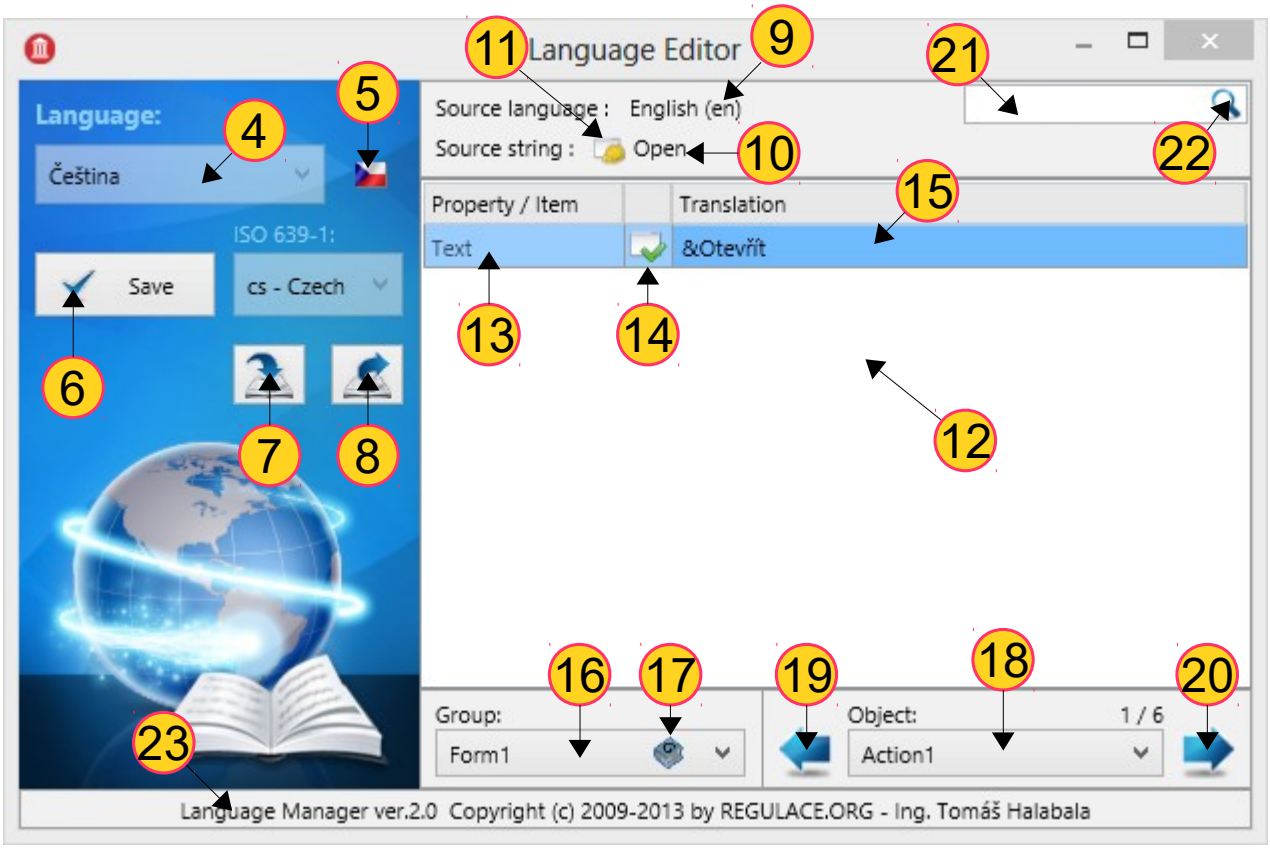


After clicking the button **(3)** you will be requested to enter the name of the language file, and subsequently this language will be created. Items for translation in the selected source language will be loaded into the editor and you can start translating.

If you launch the editor for the purposes of modifying a language and at the same time it is possible to edit more than one language, you have to first select which language you wish to open **(4)**.

(5) is a language icon. You can change this icon (flag) by clicking the left mouse button. If you wish only to delete the icon, you can click on it with the right mouse button and then confirm your decision by clicking the YES button in the displayed dialog box.

RECOMMENDATION: If you are in the VCL version of the editor you can use a simple balloon hint (**Hint**) in most of the elements. It will be displayed when you stop the mouse cursor on the control element.



FireMonkey version of the language editor

You can close the editor at any time and simultaneously save all changes by means of the button **(6)**. If you wish to exit without saving your changes, you have to close the editor by means of the cross in the right upper corner (or by means of a keyboard shortcut – in Windows it is Alt+F4) and to confirm the closing without saving the changes.

3.1.1 Import and Export

In the **Premium** and **Ultimate** editions there is also a button for importing data **(7)**.

You can import:

- Data of the same language exported by **LangMan** from any application
- A language file of the same language saved by the same application (for example for transferring from one installation into a next one)

- Data saved by the **TLang** component from the **Standard** group of the **FireMonkey** library. It is suitable during a transfer from the **TLang** component to **LangMan**.
- A general text file – every line constitutes an item in the form:

  ```
  original=translation
  ```

  where the **original** is a relevant string in the language of the design. Caution! Do not mistake the design language for the source language of the given language. The language of the design is a language in which the application is created.

Only existing and as-of-yet untranslated items of the selected language are imported from the file and an icon/flag is also imported if no icon/flag is assigned. Except text files with **$overwrite_existing=true** at the beginning of the file.

The button for exporting data **(8)** is available only in the **Ultimate** edition. A special file which can be imported in any other application with **LangMan** is created during exporting. This function saves time because you do not have to translate repeating items. The transfer of language data between different applications depends on a united design language.

3.1.2 Editing / translation

Basic information on every translated item is displayed in the upper part of the editor. Description of the source language **(9)** and text of the item in the source language. The icon **(11)** characterizes the source string in more detail. ***Table 1*** describes what each icon/colour symbolizes.

LangMan includes both the **VCL** editor for **T(Uni)LangManEngine** and the **FireMonkey** version for **TLangManEngineX**. Due to many differences between these libraries (**VCL** vs. **FireMonkey**) full uniformity of both editors is difficult to achieve. A significant difference is especially in the graphic representation of string states described in ***Table 1***.

While the **FireMonkey** editor displays states by means of icons, the **VCL** editor uses only colour rectangles. On the other hand, unlike **FireMonkey (FMX)** the **VCL** editor displays a balloon hint describing the state of the string in the textual form.

Table 1: Overview of graphical symbols characterizing language items

Representation			*LangMan Edition*	
FMX	*VCL*	*Description*	Personal Professional Premium	Ultimate
		Internal (original) language of the application design.	✓	✓
		Item loaded from a source language.	✓	✓
		Global item from a source language.	–	✓

Representation			*LangMan Edition*	
FMX	*VCL*	*Description*	Personal Professional Premium	Ultimate
		Locally translated item.	✓	✓
		Globally translated item.	–	✓
		Empty string – invalid translation.	✓	✓
		Empty global string – fill in translation.	–	✓
		Cancellation of a local translation in the given language. The text according to the source language or global translation will be used.	✓	✓
		Deletion of global translation – it applies throughout the entire application for all items which use the same global translation.	–	✓
		Keep source translation – item will remain untranslated but it will be considered translated.	✓	✓
		Restored translation – after returning a previously translated object in the form.	✓	✓
		Error – invalid translation	✓	✓

Another difference is in the switching of the item state. If, for example, any component property should not be translated in the given language on the form, you can switch it to the "**Keep source translation**" state, see Table 1. In the VCL editor you can do so by clicking on the description of the property/item **(13,14)** or by pressing the keyboard combination **Ctrl+Left arrow**, if the text cursor is located on the relevant translation **(15)**, while in the **FireMonkey** editor it is done by clicking on the state icon **(14)** of the relevant property/item **(13)**.

In the **Ultimate** edition which supports a global lexicon you can also switch items between a local translation and a translation located in the global lexicon. Normally, all translations **(15)** have a local character, which means that they are connected with the given object. If, for example, several OK buttons are present in an application, each button can be translated differently in a single language. If you, however, do not wish to make differences in a translation of several identical strings, you can save the translation in the global lexicon. When you come across a next identical item, you only need to switch to the global lexicon which will unify the translation.

IMPORTANT: It is important to realize that a substantial difference between a local and a global translation is the fact that while a local translation is bound by the address in the form Object.Property.Item (e.g. `ComboBox1.Items.Strings[1]`), a global translation is bound by the original form of the string (e.g. "`Exit`"). Therefore a global

translation is a good choice not only in repeated expressions but also in translating indexed items of a single property where their re-organization would necessarily entail subsequent modification of the order of these items also in translations in all languages!

VCL version of the language editor

Every **T(Uni)LangManEngine(X)** from the **Ultimate** edition includes within itself a common global lexicon into which other language components of **LangMan** can be connected (see the **GlobalLexicon** property). An icon **(17)** is displayed at components which have a connection to the global lexicon in the editor. If the **GlobalLexicon** property is not set to the ***true***, the items can be always translated only locally.

The current editor page **(12)** always shows only items of the currently selected group (language components) **(16)** and if this component is **T(Uni)LangManClient(X)** or **T(Uni)ResourcesTranslator(X)**, the selection is also limited to sub-objects **(18)** of the selected component. All items of a given lexicon are listed on one page in lexicons.

The **Back (19)** and **Next (20)** buttons can be used for jumping to the previous or following object. For quick searching of untranslated items you can hold the **Shift** key while clicking on one of these buttons which will lead to skipping all completely translated objects in the given direction.

3.1.3 Search

A search box **(21)** with a button **(22)** can be used for searching for strings on the current page of the editor. The search always starts from the currently selected row of the table. After reaching the last row, the search proceeds again from the beginning. Several keyboard shortcuts are associated with the

search function:

Movement of the cursor in the search box: **Ctrl+F**

Start of searching: **Enter** in the search box

F3 anywhere

Click the left mouse button on the hand glass button

Normally, only translation strings **(15)** are searched. If you wish to search also in the strings in the source language **(10)**, hold the pressed key **Ctrl** during searching. If the source language differs from the language of the application design, you can search also in the original language of the design by holding the **Shift** key.

3.1.4 Copyright panel

By setting the property of the **HideCopyright** engine to the ***true*** value, you can hide the panel **(23)** in the editor launched during the run-time of the application. This option is not accessible in the **LangMan Personal** edition.

4 LangMan Component Package Classes

4.1 TLangManEngine(X) = class (TLangManCore)

TUniLangManEngine = class (TLangManEngine)

Unit: LangManComp(X/Uni);

The main unit, further referred to as engine. It provides translations, switching between languages and manages language files.

4.1.1 T(Uni)LangManEngine(X) Methods

4.1.1.1 Function Translate (ISOorLangName: String): TLanguage;

This function translates all assigned components into required language. A parameter of type string determines the language either by name or ISO code. If the translation is realized well, an event **OnChaneLanguage** is called. After the translation, the function returns the name of the current language.

4.1.1.2 Procedure AssignLanguagesList(TargetList: TStrings);

This procedure replaces **GetLanguagesList** from the previous version of LangMan components. The procedure assign the list of loaded languages to **TargetList**. Items on this list are the valid languages and can be used as a parameter of function **Translate**.

4.1.1.3 Function GetLanguageFile(ISOorLangName: String): TFileName;

This function returns the full path to the language file specified by language name or by ISO code.

4.1.1.4 Function GetLanguageFile(LanguageIndex: Integer): TFileName;

This function returns the full path to the language file specified by item index in the list of languages, that can be obtained by **AssignLanguagesList** method.

4.1.1.5 Function GetISOCode(LanguageName: String): String;

Returns the ISO code according to the language name.

4.1.1.6 Function GetISOCode(LanguageIndex: Integer): String;

Returns the ISO code of the language specified by item index in the list of languages, that can be obtained by **AssignLanguagesList** method.

4.1.1.7 Function LangExists(ISOorLangName: String): Boolean;

Returns ***true*** if there exists the language specified by ISO code or language name.

4.1.1.8 Function FlagImage(ISOorLangName: String):TMemoryStream;

Returns **TMemoryStream** of flag image specified by language name or by ISO code.

4.1.1.9 Function FlagImage(LanguageIndex: Integer):TMemoryStream;

Returns **TMemoryStream** of flag image specified by item index in the list of languages, that can be obtained by **AssignLanguagesList** method.

4.1.1.10 Function EditorCanBeShown: Boolean;

Returns ***true*** if the language editor can be shown through **ShowLangEditor** method. The editor can be shown if exists at least one language for editing. The design language and the language from resources can not be edited, if the property **LangResourcesAllowEdit** is set to ***false***.

4.1.1.11 Procedure ShowLangEditor;

The procedure initiates a language editor. Languages in it can be only modified, but not added. A condition for starting the editor is the presence of at least one language for editing. The design language can not be edited.

Note: If you set the **LangEditorVisible** property to ***true***, the language editor will be available also in **LanguageMenu**, **TLangCombo**, **TLangFlagsCombo** and **TUniLangCombo in the VCL mode**.

4.1.1.12 Procedure ShowLangCreator;

This procedure initiates the language editor in the mode of adding new language.

Note: If you set the **LangCreatorVisible** property to ***true***, the command for the creation of a new language will be available also in **LanguageMenu**, **TLangCombo**, **TLangFlagsCombo** and **TUniLangCombo in the VCL mode**.

4.1.1.13 Procedure RefreshLangControl(LangManGUIControl: TComponent);

Call **RefreshLangControl** to refresh a visual language component such as **T(Uni)LangCombo(X)**, **T(Uni)LangFlag(X)** or **TLangFlagsCombo** specified by **LangManGUIControl** parameter.

4.1.1.14 Function ImportFromFile(FileName: String; ToCurrentLanguage: Boolean): Integer; (Premium and Ultimate Editions)

Using this function can be imported data including icon from file **FileName**. **ToCurrentLanguage** parameter determines whether they should be (***true***) data imported solely into currently selected language, or whether it can be created (if needed) (***false***) a new language. If the parameter is set to ***true*** and the language of the imported data does not match the selected language will be generated exception **EImportExport**. With this function can be imported only LangMan language files exported through language editor or through **ExportToFile** procedure or language files saved by engine of the same application.

The function **returns** the number of really imported items. This can be used both for import yet untranslated items in an existing language, as well as import entirely new language (if the parameter **ToCurrentLanguage** is set to ***false***). If the language of the imported file already exists, it will be imported only untranslated items in a given language, or will be loaded missing icon for the language, except import text file with **$overwrite_existing=true** at the beginning of the file.

Using the Import button in the language editor can be imported also lang files (**.lng**) from **TLang** component of **FireMonkey** library and simple text files that contains one translations per line in the form:

```
original=translation
```

where **original** is the string in the design language of the application.

With this import capability you can very easily and quickly migrate from other localization tools to popular **LangMan** components.

4.1.1.15 Procedure ExportToFile(FileName: String); (only in the Ultimate edition)

Export the language to file **FileName.** The exported file can be used for example to import into another project, or similar application, which can save translator time when some strings once already translated. The condition for the transmission of language data across applications is the same design language. The export can be including untranslated items - **.txt** in form `original=translation` per line, or translated items only in **LangMan** language file format – for this case recommend to choose **.llx** extension.

4.1.2 T(Uni)LangManEngine(X) Properties

4.1.2.1 Property CurrentLanguage: String

CurrentLanguage property returns the name of the currently selected language and sets the language.

4.1.2.2 Property CurrentLanguageISO: String

CurrentLanguageISO property returns the ISO code of the current language and sets the language by ISO code.

4.1.2.3 Property DesignLangISOCode: String

This property is used to set the ISO code of the designing language. **LangMan** use the two-letter language codes according to **ISO 639-1**.

4.1.2.4 Property DesignLanguageName: TLanguage

This property is used to set the name of the designing language. It means that it is the main language that is used in the design of an application. This language can be used at any time as the default language for the translation. It is therefore recommended to use the English language when designing an application, because it is most suitable language for translations into any other world language.

4.1.2.5 Property DefaultLanguage: String

The default language that is supposed to be set when starting the application. Local language or language used in the last application run can be entered into this property within **OnCreate** event of the form. The default language is loaded only at the moment of application activation, ie. when all forms are created already. Can be assigned either a language name or ISO code.

4.1.2.6 Property LangSubdirectory: String

The full path, relative path or the name of a subdirectory for storing the language files.

Windows: As a default directory is considered the directory, in which there is located the application executable file.

OSX: As a default directory is considered the user's home directory.

iOS: The default directory is the home path of the application. On iOS, it points to the device-specific location of the sandbox for the application; the iOS home location is individually defined for each application instance and for each iOS device.

Android: The default directory is the application data directory.

If this property is equal to an empty string, the language files will be stored in the default directory.

4.1.2.7 Property LangFileExtension: String

The extension of language files. In case of a multiple usage of **T(Uni)LangManEngine(X)** component in one application it is necessary to distinguish language files belonging to this component from the other language files of other engines. The language files extension is one of the appropriate options for the distinction.

4.1.2.8 Property LangFileSignature: String

The identification string of language files belonging to this engine. In case of a multiple usage of **T(Uni)LangManEngine(X)** component in one application it is necessary to distinguish language files belonging to this component from the other language files of other engines. The identification string is stored in language files, and it should distinguish language files belonging to different engines, and ideally also to different applications.

4.1.2.9 Property LangCreatorVisible: boolean

This property determines whether also the option for creating a new language should be visible in the language menu.

The language editor in **uniGUI** can be executed only in VCL mode, so this property has no effect on web forms.

4.1.2.10 Property LangEditorVisible: boolean

This property determines whether also the option for editing a language should be visible in the language menu. This option is not visible in case that is not loaded any editable language or in the instance of the web application.

The design language can not be edited. The internal languages linked to resources can be modified only if the **LangResourcesAllowEdit** property is set to ***true***.

The language editor in **uniGUI** can be executed only in VCL mode, so this property has no effect on web forms.

4.1.2.11 Property TranslateLangMan: boolean

TranslateLangMan property allows the inclusion of internal localizable strings of **LangMan** components, especially the built-in language editor, into the list of translated components.

The **LangMan** currently includes three internal languages into which it can translate itself automatically. These are English, Czech and Slovak.

4.1.2.12 Property LanguageMenu: TMenuItem

LanguageMenu property is used for the automatic generation of language

menu. Just assign any component of **TMenuItem** class from the main menu, submenu or pop-up menu.

4.1.2.13 Property LangMenuFlags: boolean

This property determines whether the flags (symbols) are supposed to be displayed in the language menu. This applies only to the menu assigned by **LanguageMenu** property.

4.1.2.14 Property DesignLangFlag: TPicture (TBitmap in X version)

Graphic symbol of the designing language. Ideally, it is the flag of the country where the language is the official language.

4.1.2.15 Property LangResources: TStringList

If you wish to have the language files as a part of the application, add them in the resources and insert the assigned ID names of these resources in this list. If you want to make it possible for these languages to be modified also later on in the run-time application editor, assign ***true*** to the **LangResourcesAllowEdit** property. Prior to the editing of the internal language, a copy is created (if it does not exist already) in the directory determined by the **LangSubdirectory** property.

4.1.2.16 Property LangFileEncoding: TLFEncoding

Language files encoding type. The default is **Unicode**. If you have any reason for a different type of language files encoding, you can choose from the following options:

type TLFEncoding = (Unicode, BigEndianUnicode, UTF8, ANSI);

Note: *This property is not available in Delphi 2007 and earlier, where the default encoding is ANSI and can not be changed.*

4.1.2.17 Property LangResourcesAllowEdit: boolean

Set to ***true*** to enable modifications and will allow overlap of the internal language files.

4.1.2.18 Property HideCopyright: boolean (Professional, Premium and Ultimate Editions)

Set to ***true*** to hide the copyright panel, which is at the bottom of the language editor. This setting does not affect on the editor running in the IDE.

4.1.3 TUniLangManEngine Properties (Premium and Ultimate Editions)

4.1.3.1 *Property CookieEnable: boolean*

Allows storage of the selected language in browser cookies for Web applications based on the framework **uniGUI**. The settings of cookies does not affect in VCL mode.

4.1.3.2 *Property CookieLifeDays: Double*

Specifies the period of cookie validity. The unit is one day, as well as the type TDateTime.

4.1.3.3 *Property CookieName: String*

Cookie name that will represent the user's selected language.

4.1.4 TLangManEngineX Properties

4.1.4.1 *Property StyleBook: TStyleBook*

The StyleBook for the language editor.

4.1.5 T(Uni)LangManEngine(X) Events

4.1.5.1 OnChangeLangQuery: TContinueQuery

The event called before the language change. If the translation into the selected language is not desirable, the translation can be stopped by setting the value ***false*** in the **Continue** return parameter.

4.1.5.2 OnChangeLanguage: TNotifyEvent

The event called after the language change.

4.1.5.3 OnBeforeEdit: TNotifyEvent

The event called before starting the language editor. When editing a language it is necessary that in the memory there is recorded the structure of all elements, that are supposed to be translated by means of the component. If for example dynamically generated forms or dynamically generated form components are supposed to be translated, it is necessary to create all these components and forms in the memory, at least temporarily, so that they could be translated. The same applies to programmable lexicon items.

4.1.5.4 OnAfterEdit: TNotifyEvent

After closing the language editor, an event **OnAfterEdit** is called. In the method, assigned to this event, components dynamically created only for the translation can be released from the memory again. Whenever the dynamic form is created again, it is immediately automatically translated into the current language. After creating the dynamic component, it is necessary to call its additional translation manually.

4.2 TLangManComponent = class (TComponent)

Unit: LangManCore;

TLangManComponent class is the basis for all other language components, such as **T(Uni)LangManClient(X)**, **T(Uni)ResourcesTranslator(X)**, **T(Uni)DesignedLexicon(X)**, **T(Uni)ProgrammableLexicon(X)** and **T(Uni)InlineLexicon(X)**.

4.2.1 TLangManComponent Properties

4.2.1.1 Property LangManEngine: TLangManCore

To the **LangManEngine** property **T(Uni)LangManEngine(X)** must be assigned. It provides the current language data to the language component.

4.2.1.2 Property GlobalLexicon: Boolean (only in the Ultimate edition)

Set to ***true*** enables for language component use the global lexicon. This property is available only in the **Ultimate Edition**, in which is incorporated into **T(Uni)LangManEngine(X)** the global lexicon. Global lexicon is common to all components connected to the appropriate engine. Into this lexicon can place arbitrary strings both lexicons and clients (forms) and then you just need translate the same string only once. For example, all the Cancel button will have in the language file only one common translation. In the language editor, the items switch to global translation individually by clicking on the button or on the status icon the left of the translated string. Click again to set back the standard local translation or the translation can be disabled for items of forms. If **GlobalLexicon** sets to ***false***, all translations may be only local and repetitive strings must be translated separately.

4.2.2 TLangManComponent Events

4.2.2.1 OnChangeLanguage: TNotifyEvent

The event called after the language change.

4.3 TLangManClient(X) = class (TLangManCPC)

TUniLangManClient = class (TLangManClient)

Unit: LangManComp(X/Uni);

This component, further referred to as the client, serves for installing the form on which it is located, into the translation by means of selected engine. In addition, it allows you to select which properties on the form and its components are supposed to be translated.

TLangManCPC = class (TLangManComponent)

Unit: LangManCore;

TLangManCPC is a cross platform ancestor of **T(Uni)LangManClient(X)** classes.

4.3.1 T(Uni)LangManClient(X) Methods

4.3.1.1 Function AddComponent (Component: TComponent; Name: string; Translate: boolean): Boolean;

If you want also form components that are created dynamically at runtime to be translated, it is necessary for each such component to be added additionally into the form component list which is maintained by the **T(Uni)LangManClient(X)** component. A condition for the proper function of automatic translation is that the form is the owner of such components. In the case of several identical dynamic components, which are to be translated consistently, you can use a common name. Then it is sufficient to register the component only once using the **AddComponent** function**.** When re-creating the same component is not required to re-register it, it is sufficient to use the same name as during the previous registration. A condition is the identical name of every dynamically created component that is supposed to be translated. Functions **AddComponent** performs the process of naming instead of you.

In the parameter **Component** must be the added component, in the parameter **Name** it is possible to specify the name of the new component. If the component already has its unique name assigned in the property **Component.Name**, use an empty strings for the parameter **Name**. The last parameter **Translate** determines whether a translation into the current language is supposed to be done immediately after adding a new component. For a multiple translation of the entire form the method **Translate** can be applied.

For an additional translation of one form component the method **TranslateComponent** can be applied.

4.3.1.2 Procedure RecreateTransStruct;

This method will rebuild the list structure of the form components. The list includes all named components, that at the given moment belong to the form,

so this method can be used for mass installing of dynamically created form components into the list of translated components. After canceling of some component this method is the only way how to remove the canceled component from the list of translated components. The presence of non-existing component in the translation list only affects the presence of this component in the language editor, which is usually rather desirable.

If a language (other than the design language) is loaded in the engine, the new components will be translated automatically.

4.3.1.3 Procedure TranslateComponent(Component: TComponent; Name: string = '');

The **TranslateComponent** method is applied for translation of one component in the currently selected language. In this process it does not matter which component is the owner of the component passed in the parameter **Component**. On the other hand, for the automatic translation it is unconditionally necessary that the form is the owner.

A condition is that the component has its unique name within the frame of the owner. In the opposite case, it is possible to specify a new name in the parameter **Name.** Otherwise, leave an empty string.

4.3.1.4 Procedure Translate;

It translates the form and all its named components. When changing the language the translation will proceed automatically, but after adding more dynamically created components, this method can be applied for the additional multiple translation of all the new named form components.

4.3.1.5 Procedure TranslateAs(FormName: String);

Specify the name of the translated form. This function is useful eg for **MDI applications.** Place **TranslateAs** to **OnCreate event** of Form and assign proper name, which will be common for all instances of the form. **LangMan** will translate all as well. String **FormName** will represent the original name of the form as component that otherwise must be unique within the application.

Note: *This method is not supported in* ***Personal Edition****!*

4.3.2 T(Uni)LangManClient(X) Properties

4.3.2.1 Property InitAfterCreateForm: boolean

This property determines whether a list of automatically translated form component is supposed to be created after the call of the method of the **OnCreate** event of the form ***true***, or even before ***false***.

4.3.2.2 Property TransAdditions: TAdditionSet

The set of names of user supplements, that are supposed to be translated.

4.3.2.3 Property TransStringProp

The set of names of properties of the type of **string** that are supposed to be translated.

4.3.2.4 Property TransTStringsProp

The set of names of properties of the type of **TStrings** that are supposed to be translated.

4.3.2.5 Property TransStructuredProp

The set of names of properties of different types, usually more complex structures, that are supposed to be translated.

4.3.2.6 Property TransOtherProp

The set of names of properties of different types, that are supposed to be translated. In some cases it may be names of the entire classes.

4.3.3 TLangManClientX Properties

4.3.3.1 Property TransXtraOptions: TFMXClientOptionsSet

A set of advanced options.

4.4 TLMList = class (TLangManComponent)

Unit: LangManCore;

The class with basic functions for language components working with a list of strings. Its descendants are **TLexicon** and **T(Uni)ResourcesTranslator(X)** classes.

4.4.1 TLMList Methods

4.4.1.1 Function IsDefined(Index: Integer): Boolean;

The function returns the ***true*** if the item on the position **Index** in the list is defined. Otherwise it returns ***false***.

4.4.1.2 Function IsDefined(LocStr: String): boolean;

This overloaded function **IsDefined** returns ***true*** if the **LocStr** in the list is defined. **LocStr** must be in design language.

4.4.1.3 Function GetLocStr(Index: Integer): String;

GetLocStr returns an item from the list at **Index** position in the design language.

4.5 TResourcesTranslator(X) = class (TCustomResourcesTranslator) TUniResourcesTranslator = class (TResourcesTranslator)

Unit: LangManComp(X/Uni);

The component **T(Uni)ResourcesTranslator(X)** is used for localizations of **resourcestrings** which have been linked to the application.

TCustomResourcesTranslator = class (TLMList)

Unit: LangManCore;

4.5.1 T(Uni)ResourcesTranslator(X)

4.5.1.1 Procedure RegisterResourceString(const Resource: String; InDesignLanguage: String = ' ');

By means of this method any string from the **resources** can be registered for localization. In the **Resource** parameter it is necessary to transfer the content of the relevant resource, NOT its name or identifier. The second parameter may be specify localized form in the language of the application design if it differs from the language in which the string is defined. As a result, immediately after calling this method, the actual content of the **Resource** string changes into the **InDesignLanguage** form, if not passed an empty string. The **T(Uni)ResourcesTranslator(X)** component can be therefore

used also for simple transcribing of any string in the **resources**. This method does not check in any way if the given resource really exists and it works even if the component is not connected to the language engine.

4.5.2 T(Uni)ResourcesTranslator(X) Properties

4.5.2.1 *Property OriginalLanguage: String*

This property has to be assigned an ISO code or a language name in which **resourcestrings** registered to localizations by means of this component are defined in the programme. If **resourcestrings** in the programme are defined in different languages, they have to be registered separately in individual **T(Uni)ResourcesTranslator(X)** components.

This is in order to use the correct versions of all registered strings in the correct languages. In your own interest do not forget to set this property correctly.

Libraries of standard **Delphi** strings are defined only for the following languages: **en** – English, **de** – German, **fr** – French and **ja** – Japanese.

4.5.2.2 *Property RegisteredStrings: Integer; (read-only)*

This property returns the total number of registered strings. During the run-time of the programme it is only for reading, but in design in the IDE it can be used to open the editor (registrator) for **resourcestrings** described in Chapter 2.3.2.

Note: If the editing button is not displayed, check if you have not forgotten to compile and install the **LangMan_Registrator** package for the platform **32-bit Windows** which is supplied together with other packages of **LangMan** components of the **Ultimate** edition. Alternatively, check out if this package is enabled in your installed components.

4.6 TLexicon = class (TLMList)

Unit: LangManCore;

The basic class of all lexicons.

4.6.1 TLexicon Methods

4.6.1.1 Function CompleteString(const Str: string): string;

The function **CompleteString** returns the given string **Str**, where the links to lexicon strings are replaced by lexicon strings in the currently selected language. These links are generated by the lexicon property **Link** and possibly **SLink**. This function is applied for translating of dynamically generated extensive texts. Instead of specific strings in a specific language use only links to specific lexicon strings while generating the text, and you will transfer the string with links into a readable form only in the moment of output to the screen, printer, etc. This function is used also by a non-visual object of **LangMan** package **TLangManStrings** (see **TLangManStrings** class description) for the dynamic translation of objects derived from **TStrings**. For example **TMemo, TRichEdit** etc.

4.6.2 TLexicon Properties

4.6.2.1 Property Link [Index: Integer]: string; (read-only)

The property **Link** returns a link of the type of string to the lexicon string to the position **Index**. A link gained in this way can be inserted into the link text during a dynamic creating of some statements, logs, etc. By means of the function **CompleteString** (see above) you can then transfer the resulting text with links into a readable form in the currently selected language. More about using this function you will find also in the non-visual **TLangManStrings** object description and in description of component **TLangManRichEdit**.

4.7 TDesignedLexicon(X) = class (TCustomDesignedLexicon) TUniDesignedLexicon = class (TDesignedLexicon)

Unit: LangManComp(X/Uni);

Except of the automatic translation of static components, located on the forms, strings for different dynamically generated dialog boxes, messages, etc. are often necessary in the programs. Lexicons are used for these purposes. Needed strings can be defined in these lexicons. These strings are then automatically translated, so the particular entry is always read in the appropriate language. Strings in **T(Uni)DesignedLexicon(X)** are defined by means of the editor directly in object inspector.

TCustomDesignedLexicon = class (TLexicon)

Unit: LangManCore;

4.7.1 T(Uni)DesignedLexicon(X) Methods

4.7.1.1 Function CreateItem(Text: string): Integer;

The function **CreateItem** is used for adding the string **Text** into the lexicon at run-time. The return value is the position (**Index**) of the new string in the lexicon.

4.7.2 T(Uni)DesignedLexicon(X) Properties

4.7.2.1 Property Item [Index: Integer]: string; (read-only)

The property **Item** returns the string from the position **Index** in the currently selected language.

4.7.2.2 Property Items: TStringList

The property **Items** is used to edit entries in the lexicon in the object inspector. It is necessary to insert these strings into the list in the design language, which corresponds to the **DesignLanguageName** of the assigned engine. In the program it is possible to read these strings in the specific selected languages by means of the property **Item**.

4.8 TProgrammableLexicon(X) = class (TCustomProgrammableLexicon) TUniProgrammableLexicon = class(TProgrammableLexicon)

Unit: LangManComp(X/Uni);

Programmable lexicon items can be defined, unlike **DesignedLexicon**, only in the program source code, for example in the **OnInitialization** event method. This lexicon is applied in cases where the language strings are known only at run-time. A condition for translating is that all items that are supposed to be translated automatically have to be in the lexicon before editing the language, otherwise it will not be possible to locate them into other languages. It follows that items of this lexicon will not be visible in language editor at design-time in Delphi IDE.

TCustomProgrammableLexicon = class (TLexicon)

Unit: LangManCore;

4.8.1 T(Uni)ProgrammableLexicon(X) Methods

4.8.1.1 Procedure DefineItem(ItemNr: Word; Text: string);

This method installs the string **Text** into the lexicon on the **ItemNr** position. If there is already a different string on this position, it will be overwritten by a new string.

4.8.2 T(Uni)ProgrammableLexicon(X) Properties

4.8.2.1 Property Item [Index: Integer]: string; (read-only)

The property **Item** returns the string from the position **Index** in the currently selected language.

4.8.3 T(Uni)ProgrammableLexicon(X) Events

4.8.3.1 OnInitialization: TNotifyEvent

This event is called when initializing the lexicon. In this event method it is possible to define lexicon items.

4.9 TInlineLexicon(X) = class (TCustomInlineLexicon)

TUniInlineLexicon = class (TInlineLexicon)

Unit: LangManComp(X/Uni); **(only in the Ultimate edition)**

You can define the items of this lexicon literally **inline** directly in the source code by means of the **Loc['**text**']** property or in the case of a link by means of the **SLink['**text**']** property. The **Inline lexicon** differs from the preceding ones in not using indexes but directly independent strings written in the source language of the design. You, however, do NOT have to initialize the items of this lexicon prior to the editing of the language (as the case is in **ProgrammableLexicon**) because they will implement themselves in the translator already during the compilation of the application. The undeniable advantage of this lexicon is of course the fact that you do not have rely on some indexes and remember which string you have under which index at all times, you only use a string directly in the source code and do not have to do anything else. The only condition is that you have to use a directly unnamed constant string designated by quotation marks in the square brackets of the **Loc** and **SLink** properties. It is not possible to transfer a variable or an indirectly defined constant. In such cases initialization of these entries would be necessary before language editing.

TCustomInlineLexicon = class (TLexicon)

Unit: TLangManCore;

4.9.1 T(Uni)InlineLexicon(X) Properties

4.9.1.1 Property Item [Index: Integer]: string; (read-only)

The property **Item** returns the string from the position **Index** in the currently selected language.

4.9.1.2 Property Loc [LocalizableString: String]: string; (read-only)

The property **Loc** returns passed **LocalizableString** in the currently selected language. As mentioned, the **LocalizableString** must be defined directly by using quotation marks in the design language in the form **Loc['**any text in the design language**']**.

4.9.1.3 Property SLink [LocalizableString: String]: string; (read-only)

The property **SLink** returns a link of the type of string to the localized **LocalizableString** in lexicon. A link gained in this way can be inserted into the link text during a dynamic creating of some statements, logs, etc. By means of the function **CompleteString** (see chapter 4.6.1.1) you can then transfer the resulting text with links into a readable form in the currently selected language. For more information on this feature can be found in the description of non-visual object **TLangManStrings** or in description of component **TLangManRichEdit** with which can be used **SLink** too.

4.10 TLangCombo(X) = class (TCustomComboBox)

TUniLangCombo = class (TUniCustomComboBox)

Unit: LangManComp(X/Uni);

This is a standard **ComboBox**, which is after assigning to the engine automatically loaded with existing languages. All clients and lexicons will be translated when selecting a language.

4.10.1 T(Uni)LangCombo(X) Properties

4.10.1.1 Property LangManEngine: T(Uni)LangManEngine(X)

To this property must be assigned the adequate engine, with which the combo box to be connected.

4.10.2 T(Uni)LangCombo Properties

4.10.2.1 Property StyleCombo: TLangComboStyle

This property is analogous to **Style ComboBox** property. The difference is that here it is not possible to set the **csDropDown** style.

4.10.3 T(Uni)LangCombo(X) Events

4.10.3.1 OnChangeLanguage: TNotifyEvent

The event called after the language change.

4.11 TLangFlagsCombo = class (TCustomComboBoxEx)

Unit: LangManComp;

LangFlagsCombo is an enhanced **ComboBox** that has also relevant flags (languages icons) displayed in front of the names of languages. The function is similar to **TLangCombo**.

4.11.1 TLangFlagsCombo Properties

4.11.1.1 Property LangManEngine: TLangManEngine

The **TLangManEngine** must be assigned to this property.

4.11.2 TLangFlagsCombo Events

4.11.2.1 OnChangeLanguage: TNotifyEvent

An event called after the language change.

4.12 TLangFlag(X) = class (TImage)

TUniLangFlag = class (TUniImage)

Unit: TLangManComp(X/Uni);

This visual component is almost identical to the **TImage (TUniImage)**. Additionally includes property **LangManEngine** for links with the engine whose flag icon of the currently selected language to be displayed.

4.12.1 T(Uni)LangFlag(X) Properties

4.12.1.1 Property LangManEngine: T(Uni)LangManEngine(X)

To this property must be assigned the adequate engine, with which the **T(Uni)LangFlag(X)** to be connected.

4.13 TValuedLabel = class (TCustomLabel)

TUniValuedLabel = class (TUniLabel)

TValuedLabelX = class (TLabel)

Unit: LangManCtrls(X), LangManCompUni;

T(Uni)ValuedLabel(X) are an additional components similar to the standard **TLabel (TUniLabel)**. It has **ValueName**, **ValueSeparator**, **ValueSpaces** and **Value** properties instead of the **Caption (Text)** property. These properties are linked in the final display, while **ValueSpaces** indicates the number of spaces instead of **ValueSeparator**. The trick is that **LangMan** translates (except **Hint** on **TValuedLabel**) only the **ValueName** property. It is sufficient to enter only the **Value** in the program.

The resulting string that is displayed by this visual component has the following format: '**ValueName**' + '**ValueSeparator**'+ **ValueSpaces x ' ' +** '**Value**'.

4.13.1 T(Uni)ValuedLabel(X) Properties

4.13.1.1 Property Value: TCaption

Any string can be assigned to this property.

4.13.1.2 Property ValueName: TCaption

Into the translations only this property is included to which the name of the **Value** in the designing language should be assigned.

4.13.1.3 Property ValueSeparator : string

Any string can be assigned to this property.

4.13.1.4 Property ValueSpaces : byte

This property indicates the number of spaces between **ValueSeparator** and **Value**.

4.14 TLangManStrings = class (TStringList)

Unit: LangManCore;

TLangManStrings class serves for an easy dynamic translation of objects derived from the **TStrings** class – from various lists to the entire extensive text files, visual components **TMemo**, **TRichEdit** etc. In the moment when you enter in the program for example a log into a component **TMemo** and you want this statement, created at runtime ,to be translated into the selected language after the change of a language, **TlangManStrings** will take care of it instead of you.

4.14.1 TLangManStrings Constructor

4.14.1.1 Constructor Create(ControlledStrings: TStrings; Lexicon: TLexicon);

When creating an object of the **TLangManStrings** class, it is necessary to transfer the following in the constructor parameters:

In **ControlledStrings** an object derived from **TStrings** must be transferred. It is supposed to be managed by the new object – from a functional point of view it the replacement of the original object by the newly created object. The original object, of course, still exists, but it only serves for the output of the translated text in the selected language, while the new object only operates with the links to the lexicon strings.

In the **Lexicon** parameter a lexicon must be transferred. The strings of this lexicon are supposed to be used for the texts translation.

From the moment of calling this constructor the text content of the original object of the type of **TStrings** should be in the program treated only by means of this object, which is derived from the **TStringList** class, and is therefore also the **TStrings** class descendant.

Its methods **Add, AddObject, Insert, InsertObject, Delete, Clear, Exchange, Sort**, **CustomSort** have the same function and the same effect on the original object as if it were the methods of the object itself.

4.14.2 TLangManStrings Methods

4.14.2.1 Procedure Translate;

This procedure will re-translate the entire text in the managed object. This procedure is called automatically when changing the lexicon language.

4.15 TLangManRichEdit = class (TCustomRichEdit)

Unit: LangManComp;

It is a visual component developed for the same purpose as the **TLangManStrings** class. For example, to generate a variety of listings, logs, reports, etc. but here the additional capability to format the text. You can insert strings with different fonts, different character sizes, colors, styles, etc. with the language of the final document corresponds to the currently selected language and the entire document is automatically translated with changing the language.

The property **ReadOnly** is hidden and set to ***true***. To write and edit text content are in **TLangManRichEdit** implemented the necessary methods, described bellow.

4.15.1 TLangManRichEdit Methods

4.15.1.1 Procedure AssignStyles(LMStringStyles: TLMStringStyles);

Can be used for direct assignment array of type **TLMStringStyles** with definitions of styles for your document. **TLMStringStyles** type is defined as follows:

```
TLMStringStyles = array of TLMStringStyle;
```

where

```
TLMStringStyle = record
    Color: TColor;
    Font: TFontName;
    Charset: TFontCharset;
    Style: TFontStyles;
    Size: Integer;
    Pitch: TFontPitch;
end;
```

4.15.1.2 Procedure ClearStyles;

Clears the array of styles definitions.

4.15.1.3 Function GetStyles: TLMStringStyles;

Returns an array of font style definitions.

4.15.1.4 Function StylesCount: Integer;

Returns the number of defined styles.

4.15.1.5 Function SetStyle(Style: TFontStyles; Size: Integer = 0; Color: TColor = clDefault; FontName: TFontName = ''; Charset: TFontCharset = DEFAULT_CHARSET; Pitch: TFontPitch = fpDefault; StyleIndex: ShortInt = -1): Integer;

Create or modify the font style. Each font style based on the **Font** property. Except the first argument **Style** are all other arguments optionally. Default value indicate that it applies appropriate value of the **Font** property. The last parameter is the style index. Default value **-1** means adding a new style at the end of the array of styles.

SetStyle returns index of the created/modified font style.

4.15.1.6 Function Format(const Text: String; StyleIndex: ShortInt): String;

This function formats the **Text** according to the preset style specified by **StyleIndex**. The result of this function can be used to write into the document by using **Write**, **WriteLn**, **RewriteLine** or **InsertLine** methods.

4.15.1.7 Procedure Write(const Text: String; StyleIndex: ShortInt = -1);

Writes a **Text** at the end of the document. If you pass a parameter **StyleIndex** will be used this style on the unformatted parts of the **Text**. Any partial sections of **Text** can be formatted by other styles using **Format**.

Example

```
With LangManRichEdit1 do
Write('Hello '+Format('whole '+Format('world',3)+' !',2),1);
```

The result may look like this: Hello ***whole*** *world* ***!***

4.15.1.8 Procedure WriteLn(const Text: String; StyleIndex: ShortInt = -1);

Same as **Write**, however the **Text** is terminated by line break.

4.15.1.9 Procedure NextLine;

Wraps the current line.

4.15.1.10 Procedure Clear;

Deletes the entire document.

4.15.1.11 Function LinesCount: Integer;

Returns the number of rows in the document. Automatic word wrap if **WordWrap** is set to ***true***, has no influence the **LinesCount**.

4.15.1.12 Function ReadLineText(LineIndex: Integer): String;

Returns line specified by **LineIndex**. The first line corresponds with **LineIndex** = 0. Automatically wrapped line is returned complete and is counted as one line.

4.15.1.13 Function ReadLineFText(LineIndex: Integer): String;

Returns line specified by **LineIndex** including formatting tags and lexicon links. These data are suitable for modification and subsequent return to the line by **RewriteLine**.

4.15.1.14 Procedure DeleteLine(LineIndex: Integer);

Deletes the line by **LineIndex**. The first line corresponds with **LineIndex** = 0.

4.15.1.15 procedure RewriteLine(LineIndex: Integer; const Text: String; StyleIndex: ShortInt = -1);

Replace the line with a new **Text**. You can define a style for the line, as well as **Write**, **WriteLn** and **InsertLine** methods.

4.15.1.16 Procedure InsertLine(LineIndex: Integer; const Text: String; StyleIndex: ShortInt = -1);

Inserts **Text** at **LineIndex** in document. You can define a style for the line, as well as **Write**, **WriteLn** and **RewriteLine** methods.

4.15.1.17 Procedure Translate;

Translates the entire document into the currently selected language. This method is called automatically when changing the language of the appropriate **TLangManEngine**.

4.15.1.18 Procedure LoadFromFile(const SourceFile: TFileName; Encoding: TEncoding);

Call **LoadFromFile** to fill the document from the file specified by **SourceFile**. This file may contain formatting tags and lexicon links, so that the result document is displayed in the selected language.

If the **Encoding** parameter is not given, then the file is loaded using the appropriate encoding.

4.15.1.19 Procedure LoadFromStream(SourceStream: TStream; Encoding: TEncoding);

Call **LoadFromStream** to fill the document from the stream specified by **SourceStream**. This stream may contain formatting tags and lexicon links, so that the result document is displayed in the selected language.

If the **Encoding** parameter is not given, then the stream is read using the

appropriate encoding.

4.15.1.20 Procedure SaveRichTextToFile(const DestinationFile: TFileName; Encoding: TEncoding);

Call **SaveRichTextToFile** to save the document to the RTF file specified by **DestinationFile**. If the **Encoding** parameter is not given, then the document is saved with the encoding specified in the **Encoding** property.

4.15.1.21 Procedure SaveRichTextToStream(DestinationStream: TStream; Encoding: TEncoding);

Call **SaveRichTextToStream** to save the document to the stream specified by **DestinationStream**. The document is saved in RTF format. If the **Encoding** parameter is not given, then the document is saved with the encoding specified in the **Encoding** property.

4.15.1.22 Procedure SaveEncodedFormToFile(const DestinationFile: TFileName; Encoding: TEncoding);

Saves the document including formatting tags and lexicon links to the file specified by **DestinationFile**. The file saved by this method, you can load again using the **LoadFromFile** with maintained functionality of automatic translations.

4.15.1.23 Procedure SaveEncodedFormToStream(DestinationStream: TStream; Encoding: TEncoding);

Saves the document including formatting tags and lexicon links to the stream specified by **DestinationStream**. The stream saved by this method, you can load again using the **LoadFromStream** with maintained functionality of automatic translations.

4.15.2 TLangManRichEdit Properties

4.15.2.1 Property AssignedLexicon: TLexicon

To this property must be assigned a lexicon, whose string items are used to create the localizable document. Strings inserted to the document by using the **Link** or **SLink*** will be automatically inserted in the selected language and will be automatically translated when changing the language of the appropriate **TLangManEngine**.

4.15.2.2 Property AutoFont: Boolean

Set this property to ***false*** to disable automatic font change. Standard **TRichEdit** component causing undesirable font change on some accented characters even if it is not needed. Therefore, to the **TLangManRichEdit**

* **SLink** is supported only in **Ultimate Edition**

component was added this property named **AutoFont**, that allows easily disable of this mentioned feature.

4.15.2.3 Property Link [Index: Integer]: string; (read-only)

The property **Link** returns a link of the type of string to the lexicon item specified by **Index**. When insert the link into the document prints out the item of lexicon in the currently selected language and after switching the language all so embedded sections of text in entire document is automatically overwritten in the new language.

The following example demonstrates how to insert item No.1 of associated lexicon. A prerequisite is correctly set the **AssignedLexicon** property.

```
With LangManRichEdit1 do Write(Link[1]);
```

4.15.2.4 Property SLink [LocalizableString: String]: string; (read-only) (only in the Ultimate edition)

The property **SLink** returns a link of the type of string to the localized **LocalizableString** in lexicon. When insert the link into the document prints out the **LocalizableString** of lexicon in the currently selected language and after switching the language all so embedded sections of text in entire document is automatically overwritten in the new language.

The following example demonstrates how to insert into the document the localizable string 'Hello world!'. A prerequisite is correctly assigned any **TInlineLexicon** to the **AssignedLexicon** property.

```
With LangManRichEdit1 do Write(SLink['Hello world!']);
```

Note: It is not essential if the string, in this case, 'Hello world!', was earlier defined in the lexicon, because the **SLink** property it also defines as well as **Loc** property of appropriate **TInlineLexicon**.

4.16 TLangFlagSelectorX = class (TScrollBox)

Unit: LangManCompX;

TLangFlagSelectorX is a scrollbox with the flags buttons of all available languages. After clicking on the flag will be switched the application to the appropriate language. This component does not require any programming. Allows horizontal and vertical orientation with adjustable flag spacing. On **TLangFlagSelectorX** can be placed any effect component. This effect is automatically applied to the flag of the currently selected language.

4.16.1 TLangFlagSelectorX Properties

4.16.1.1 Property LangManEngine: TLangManEngineX

To this property must be assigned the adequate engine, with which the **TLangFlagSelectorX** to be connected.

4.16.1.2 Property LangManGridOrientation: TLMOrientation

foHorizontal determines the arrangement of flags in the row and **foVertical** option determines the vertical arrangement in the colum.

4.16.1.3 Property LangManGridHorzAlign: TLMFlagsGridAlign

This property specifies **Leading**, **Center** or **Trailing** horizontal align of flags within the component area.

4.16.1.4 Property LangManGridVertAlign: TLMFlagsGridAlign

This property specifies **Leading**, **Center** or **Trailing** vertical align of flags within the component area.

4.16.1.5 Property LangManGridMargin: Single

Specifies the offset of the first or the last flag from the edge of the box.

4.16.1.6 Property LangManGridGaps: Single

Specifies the space width between the flags buttons.

4.16.1.7 Property LangManFlagsHeight: Single

Specifies the height of all flags buttons.

4.16.1.8 Property LangManFlagsWidth: Single

Specifies the width of all flags buttons.

4.16.1.9 Property LangManFlagsWrapMode: TImageWrapMode

Specifies whether and how to resize, replicate, and position the flag image for flags rendering. See **TImage.WrapMode**.

4.16.2 TLangFlagSelectorX Events

4.16.2.1 OnSelectLanguage: TNotifyEvent

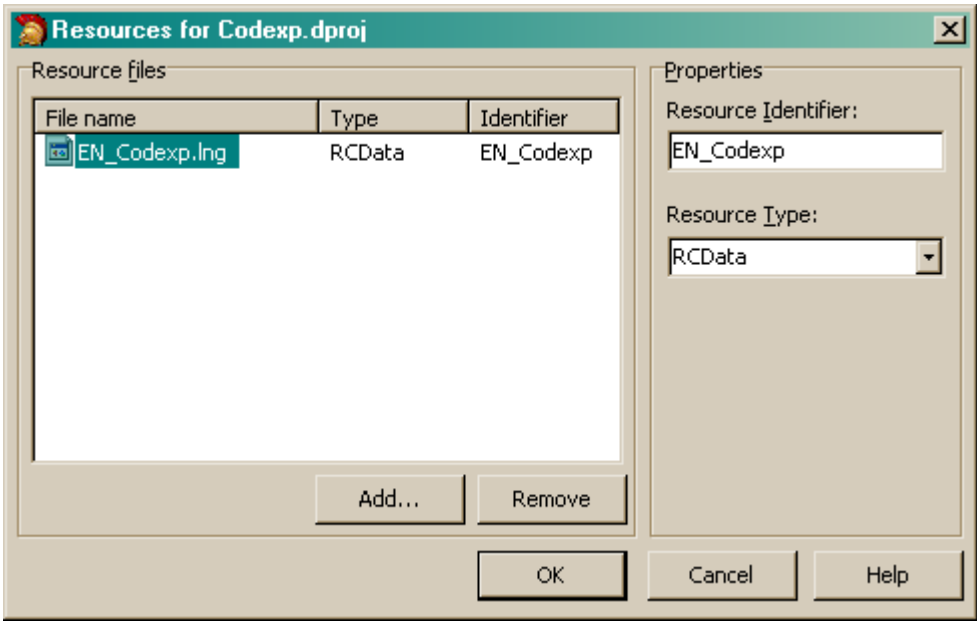
An event called after the language change due click on the flag button.

5 Language files as application resources

The **T(Uni)LangManEngine(X)** normally looks the language files according to the rules defined by properties **LangFileExtension**, **LangFileSignature** and **LangSubdirectory** on file system of device / computer. From application resources read only files specified in **LangResources** (see chapter 4.1.2.15).

Below I will describe how insert language files into application resources.

In the Delphi IDE from menu **Project / Resources**... open a dialog box for editing the resources, which are to be linked into the application file.



Obr. 5.1: Dialog for resources administration

Use the **Add** button to insert a valid language file and in the field **Resource Identifier** enter your own name to the added language. Choose this name aptly if possible and realize that in case of editing the internal language by the user, this language will be exported to the disk under the same name and with an extension assigned according to **LangFileExtension** property of the appropriate language engine. Keep the **Resource Type** on **RCData.**

Once you click on **OK**, and create the program (Build) all selected language files will be linked to file of your application in the described manner. During every other program creating the actual content of language files will be used, so you do not have worry about the constant importing of language files after every change.

Nothing changes on the manner of creating and modifying the language files. You can use the language editor **LangMan** at any time and you can rely on the fact that during the following compilation will be linked always the current version.

Now it is still necessary to order to the relevant language engine

(**T(Uni)LangManEngine(X)**) into the **LangResources** property which internal languages from the resources are supposed to be loaded. After opening the **LangResources** in object inspector, the **String List Editor** will be displayed. On the individual lines now enter the names of internal language files (**Resource Identifier**) that you assigned in dialog box **Resources** to individual language files.

6 Dynamic generating of texts

If your program generates a statement, a log file, etc. at run-time, it can insert into the file relevant strings from the lexicon, which lists them in the currently selected language. However, this approach has one weakness. After the following language change, the existing content of the statement stays in the original language, while the continuing statement will be in the changed language. Under normal circumstances, there is no reasons for the user to select a different language than the one selected in the beginning and the most suitable one. There are some cases when the change of a language is desirable at run-time of the program, and it is even desirable to change the language of the entire statement from the beginning not only for the following part.

For these cases, a new class **TLangManStrings** and visual component **TLangManRichEdit** (described in chapter 2.9 and 4.15) has been created and into the lexicons the method **CompleteString** and **Link** and **SLink** properties has been added.

Example of using class TLangManStrings with TMemo component:

If you have in your application inserted visual component **TMemo,** into which the program generates some statement at run-time, and you need the language of the entire statement to be possible to be changed at any time, use the **TLangManStrings** class. The procedure is as follows:

In the routine of the method of the **OnCreate** event of the form you will create an object for instance Memo:

```
Memo := TLangManStrings.Create(Memo1.Lines, MainLexicon);
```

where **Memo1** is the visual component **TMemo,** and **MainLexicon** is a language lexicon (**T(Uni)DesignedLexicon(X)**, **T(Uni)ProgrammableLexicon(X)** or **TInlineLexicon(X)**), which contains all the necessary language strings for the dynamic generation of your statement (it is not necessary for **T(Uni)InlineLexicon(X)**).

This step can be regarded as a replacement of **Memo1.Lines** property by a new object **Memo**. So from this moment on you must change all the text content of **Lines** property solely by means of the new object **Memo** of **TlangManStrings** class! Any manipulation of the strings in the **Memo1.Lines** property will be done solely by means of the object **Memo**, and it will be done in the very same way as if you were working directly with **Memo1.Lines**. For example, a new line will be inserted as follows:

```
Memo.Add('Text of a new line');
```

The result will be adding of the string **'Text of a new line'** on a new line of the **Memo1.Lines** property.

Now we come to creating a text translatable by **MainLexicon** lexicon, which was in the **Memo** object constructor passed as the second parameter. So for example if there is a string '**My Text'** (current language is the English) on the item with **index 2** of the lexicon **MainLexicon,** you could insert the string directly in the following way:

```
Memo.Add(MainLexicon.Item[2]);
```

The effect would be the same as in this case:

```
Memo.Add('My Text');
```

The only advantage would be that the just inserted string would be in the currently selected language. In the moment, when the user changed the language of the appropriate engine, the inserted text would stay unchanged.

However, if you insert the string by means of the link (**Link** or **SLink**):

```
Memo.Add(Memo.Link[2]);            // if MainLexicon is not Inline Lexicon
Memo.Add(Memo.SLink['My text']); // if MainLexicon is Inline Lexicon
```

in the given moment in the **Memo1** component the same text would be on the line as in previous example, but with the difference that, if the user changed the language, the text on the last line of **Memo1.Lines** would be automatically translated into the newly selected language.

In the same way as the method **Add** work without any difference from the original methods of the **TStrings** class also other methods such as **AddObject, Insert, InsertObject, Delete, Clear, Exchange**, **Sort** and **CustomSort** of the **TLangManStrings** class. It is only necessary to pay attention to the fact that the resulting assigning of indexes to strings of the **Lines** property of the **TMemo** component may differ from the object of **TLangManStrings** class. For example, if you have in the component **TMemo** set an automatic line wrap, the number of lines may increase by the number of wrapping compared to the number of lines in **TLangManStrings**. This is one more important reason why to follow the rule to approach the original text of the object only by means of the compensatory object **TLangManStrings**.

The reward is a very impressive on-time independent possibility of changing the language of extensive **TStrings** texts and again, as it is usual in case of **LangMan** components, without any demands for programmer's time. Otherwise, he would have to use much more complicated and laborious way to achieve the same result.

7 Tips and Tricks

7.1 Fast editing of items of the DesignedLexicon

By double clicking on the **T(Uni)DesignedLexicon(X)** component in IDE Delphi you will easily open the item editor of the **Items** property.

7.2 Inserting special characters

LangMan allows the use of all characters of the Unicode set (from Delphi 2009) including special characters with a numerical value lower than 32. If the required character cannot be directly entered in the editor by means of the keyboard, you can enter any character (except zero) by means of a substitute code in the following way:

```
'#code'
```

where **code** is the decimal code of the required character.

Concatenation of several special characters following each other is also allowed. For example, the code for a new line may look as follows:

```
'#13#10'
```

A single quotation mark at least in front of the first character **#** is very important.

If you need a string to include a formula in the form **'#number**, without the relevant character being automatically inserted, replace **#** with substitution code **'#35'** . For example “xx'#1” can be written as follows:

```
xx''#35'1
```

7.3 LangMan integration on a form at run-time

The **T(Uni)LangManClient(X)** component can be inserted and activated even during the run-time of a programme on any form (**TForm**, **TFrame**, **TDataModule**) or also on any component of the **TComponent** class or a descendant of this class. This is suitable especially in forms of third parties in whose code we cannot modify, for example, because we do not have source codes.

For example, for localization of the **Form** by the **LangMan** engine it suffices to call:

```
TLangManClient.Create(Form).LangManEngine := LangMan;
```

In the first, the **TLangManClient** component is created, and it is subsequently inserted in the **Form** and in the end the client is activated by assigning the existing **LangMan** engine. If the original client setting is not suitable and you wish, for example, to allow the global lexicon and change the selection of properties which should be localized, it is better to do this before assigning the engine, for example, in the following way:

```
with TLangManClient.Create(Form) do begin
  GlobalLexicon := true;
  Name := 'LMClient';
  TransStringProp := TransStringProp - [pnText];
  LangManEngine := LangMan;  // activate
end;
```

7.4 Unsuitable default configuration of projects

In the latest **Delphi**, an independent target directory is selected in the default project settings for building an application in the **Debug** configuration and an independent directory for the **Release** configuration. In many cases this division has its advantages and sometimes it can be preferred also with the use of the **LangMan** components. You have to realize, however, that in the case of this setting a possible relative path (even an empty string is a relative path) set in the **LangSubdirectory** property of the engine will generate different locations also for language files in the individual configurations. Sometimes it is enough to have the current language files, for example, only in the **Release** configuration and to update them in other configurations only sometimes or not at all. However, it will be often desirable (at least to maintain order) for the project to use the same language files in all configurations. If this is also your case, there are several solutions from which you can choose:

- to set the same target directory for all configurations, or
- to use the absolute path in **LangSubdirectory,** or
- using of a conditional compilation to rewrite the path in **LangSubdirectory** in non-**Release** configuration, or
- use only one project configuration and to change it according to the current needs, or
- using of a conditional compilation to allow the language editor only in a single configuration, and/or
- synchronize the language files after every modification.

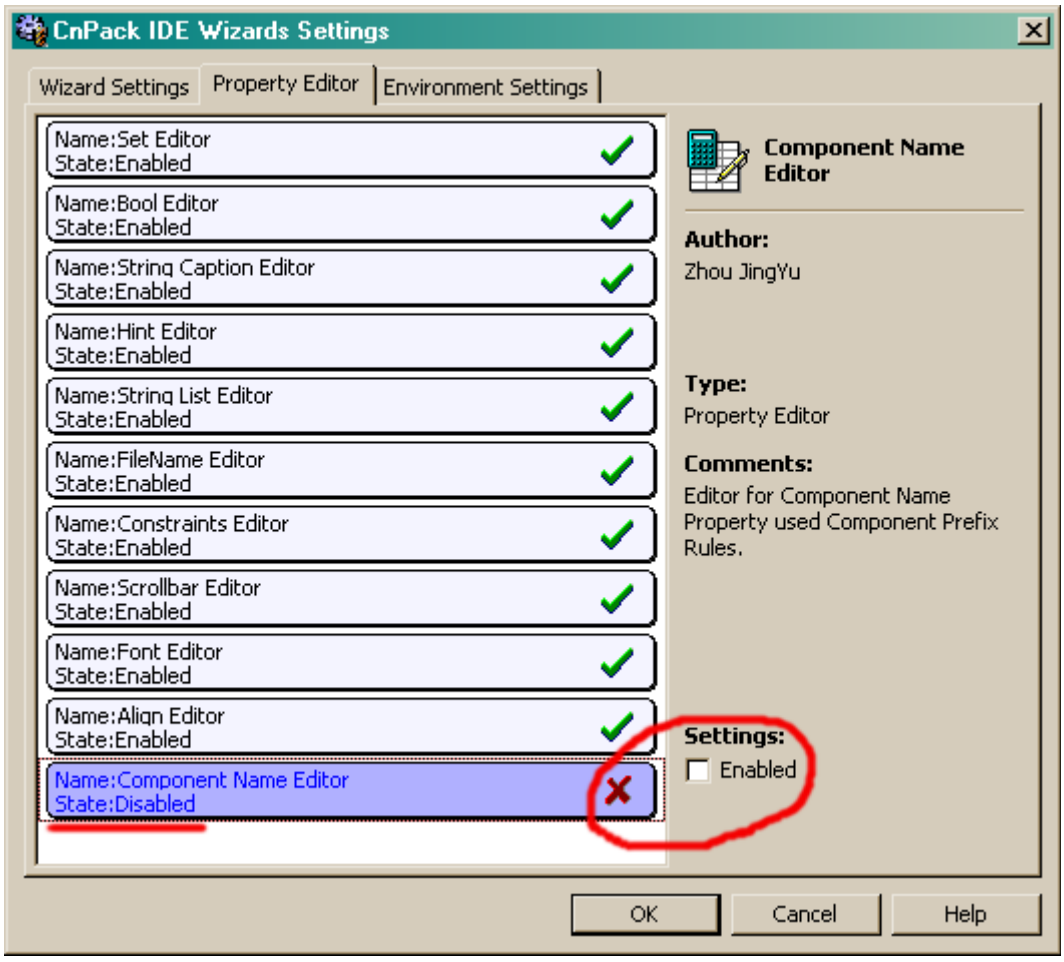
7.5 Component Name Editor

Notice that in language files data are often recorded with the names of individual components to which the relevant strings belong. When the name of a localized component is changed, it may be necessary to modify in a corresponding way also the language files. **LangMan** can perform these modifications automatically if a localized component is renamed and uses for this purpose its own **Property Editor** for **Name** properties of the **TComponentName** type.

There are, however, several other tools of third parties which you can install in **Delphi** and which also includes component name editor. Unfortunately, only one editor can be active for the same property which means that if any other editor were to replace the **LangMan** editor, automatic corrections of language files would not work when components are renamed.

I recommend that you do not use an alternative component name editor so that you can avoid unnecessary problems.

For example, the tool package **CnPack**, which includes such an editor, is very widespread. Therefore I recommend that you switch off this function in the setting of **CnPack**. It is enough to uncheck the **Enable** box at the **Component Name Editor** row in the **CnPack** setting on the **Property Editor** bookmark (see the figure). The performed change of the setting will remain valid until the next start of **Delphi (RAD Studio)**.



7.6 After upgrading always recompile

LangMan components package is necessary rebuild after each upgrading **Delphi**, **hotfixes** or after upgrade any libraries, **TeeChart**, **uniGUI framework**, etc.

8 Product comparison

Feature	LangMan Personal	LangMan Professional	LangMan Premium	LangMan Ultimate
Run-time language editor	✓	✓	✓	✓
Design-time language editor	–	–	✓	✓
Multiple engines per project	✓	✓	✓	✓
Supporting MDI apps	–	✓	✓	✓
Supporting web apps (uniGUI)	–	–	✓	✓
Import	–	–	✓	✓
Export	–	–	–	✓
Global Lexicon	–	–	–	✓
Support run-time translations of **RESOURCESTRINGs**	–	–	–	✓
Components				
TLangManEngine	✓	✓	✓	✓
TUniLangManEngine	–	–	✓	✓
TLangManEngineX	–	–	✓	✓
TLangManClient	✓	✓	✓	✓
TUniLangManClient	–	–	✓	✓
TLangManClientX	–	–	✓	✓
TResourcesTranslator	–	–	–	✓
TUniResourcesTranslator	–	–	–	✓
TResourcesTranslatorX	–	–	–	✓
TDesignedLexicon	✓	✓	✓	✓
TUniDesignedLexicon	–	–	✓	✓
TDesignedLexiconX	–	–	✓	✓
TProgrammableLexicon	✓	✓	✓	✓
TUniProgrammableLexicon	–	–	✓	✓
TProgrammableLexiconX	–	–	✓	✓
TInlineLexicon	–	–	–	✓
TUniInlineLexicon	–	–	–	✓
TInlineLexiconX	–	–	–	✓
TLangCombo	✓	✓	✓	✓
TUniLangCombo	–	–	✓	✓

Feature	LangMan Personal	LangMan Professional	LangMan Premium	LangMan Ultimate
TLangComboX	–	–	✓	✓
TLangFlag	✓	✓	✓	✓
TUniLangFlag	–	–	✓	✓
TLangFlagX	–	–	✓	✓
TLangFlagsCombo	✓	✓	✓	✓
TLangFlagSelectorX	–	–	✓	✓
TValuedLabel	✓	✓	✓	✓
TUniValuedLabel	–	–	✓	✓
TValuedLabelX	–	–	✓	✓
TLangManRichEdit	✓	✓	✓	✓
TLangManStrings class	✓	✓	✓	✓
Sets of components supported by LangMan Clients				
FireMonkey (FMX)	–	–	✓	✓
VCL (except Ribbon Controls)	✓	✓	✓	✓
Ribbon Controls (TResourcesTranslator required)	–	–	–	✓
Rave Reports	✓	✓	✓	✓
TeeChart	✓	✓	✓	✓
SynEdit	✓	✓	✓	✓
uniGUI	–	–	✓	✓
Can be adding other components	✓	✓	✓	✓
Supported platforms				
32-bit Windows	✓	✓	✓	✓
64-bit Windows	✓	✓	✓	✓
OS X	–	–	✓	✓
iOS	–	–	✓	✓
Android	–	–	✓	✓
Support for the Microsoft style Modern UI (previously Metro UI)				
VCL Metro	✓	✓	✓	✓
FireMonkey (FMX) Metro	–	–	✓	✓
Other advantages				
Include source codes	✓	✓	✓	✓
Allowed modifications	–	–	–	✓
Commercial purposes	–	✓	✓	✓

Feature	LangMan Personal	LangMan Professional	LangMan Premium	LangMan Ultimate
Technical Support	–	★	★★★	★★★★★

9 Limitations according to platforms

Some functions of the **LangMan** components depend on the platform, and they therefore do not work in all systems. The following table shows these functions and what their restrictions are.

Feature / component	Platform				
	32-bit Windows	64-bit Windows	OSX	iOS	Android
Built-in Language Editor	✓	✓	✓	–	–
T(Uni)ResourcesTranslator(X)	✓	✓	–	–	–

Document Revision / History

Date	Revision	Note
4.10.2015	13	LangMan 2.6.2 Version Release + Support Delphi 10 Seattle
12.4.2015	12	LangMan 2.6.0 Version Release + Support Delphi XE8
22.1.2015	11	LangMan 2.5.0 Version Release + New visual component TLangFlagSelectrorX
8.9.2014	10	LangMan 2.2.0 Version Release + Support Delphi XE7
17.4.2014	09	LangMan 2.1.1 Version Release + Support Delphi XE6 + Italian internal language + Export/Import – more skills
9.12.2013	08	New Tips and Tricks
6.12.2013	07	LangMan 2.0.0 Version Release + Run-time translations of RESOURCESTRINGs + FireMonkey support + Support 64-bit Windows + Support OS X + Support iOS + Support Android + Support uniGUI framework + Support SynEdit components + Full support of Ribbon Controls + Global lexicon + TLangFlag(X) + TUniLangFlag + TUniLangCombo + TUniValuedLabel + TInlineLexicon(X) + TResourcesTranslator(X) + Search in strings + Possible use of the control characters + Blocking translation by using Tag property + Design-time editor + Import / export + ISO codes
9.1.2013	06	LangMan 1.2.4 Version Release + MDI support
11.2.2012	05	LangMan 1.2.1 Version Release
24.1.2012	04	LangMan 1.2.0 Version Release + TLangManRichEdit
15.4.2011	03	LangMan 1.1.8 Version Release

15. 8. 2010	02	LangMan 1.1.1 Version Release + TLangManStrings
17.2.2010	01	LangMan 1.1.0 Version Release + language files in resources

Information About the Producer

Ing. Tomáš Halabala – REGULACE.ORG
Slunná 848, Luhačovice
CZ-76326
Czech Republic

Tel.: **+420 728 677 659**
E-mail: **info@regulace.org**
Web: **http://www.regulace.org**