

LangMan 1.2

The package of localization components for Delphi

(support UNICODE)

MANUAL



REGULACE.ORG

HW & SW Development



OBSAH

1 General Information.....	4
1.1 Supported versions of Delphi.....	5
1.2 Install LangMan components.....	5
2 LangMan 1.2 Package Components.....	8
2.1 TLangManEngine.....	8
2.2 TLangManClient.....	8
2.3 TDesignedLexicon.....	8
2.4 TProgrammableLexicon.....	8
2.5 TLangCombo.....	9
2.6 TLangFlagsCombo.....	9
2.7 TValuedLabel.....	9
3 LangMan Component Package Classes.....	10
3.1 TLangManEngine = class (TComponent).....	10
3.1.1 TLangManEngine Methods.....	10
3.1.1.1 Function Translate (LangName: TLanguage): TLanguage;.....	10
3.1.1.2 Function GetLanguagesList: TStrings;.....	10
3.1.1.3 Function GetLangFilesList: TStrings;.....	10
3.1.1.4 Procedure ShowLangEditor;.....	10
3.1.1.5 Procedure ShowLangCreator;.....	10
3.1.2 TLangManEngine Properties.....	11
3.1.2.1 Property CurrentLanguage: String; (read-only).....	11
3.1.2.2 Property DesignLanguageName: TLanguage;	11
3.1.2.3 Property DefaultLanguage: TLanguage;	11
3.1.2.4 Property LangSubdirectory: String;	11
3.1.2.5 Property LangFileExtension: String;	11
3.1.2.6 Property LangFileSignature: String;	11
3.1.2.7 Property LangCreatorVisible: boolean;	12
3.1.2.8 Property LangEditorVisible: boolean;	12
3.1.2.9 Property TranslateLangMan: boolean;.....	12
3.1.2.10 Property LanguageMenu: TMenuItem;	12
3.1.2.11 Property LangMenuFlags: boolean;	12
3.1.2.12 Property DesignLangFlag: TPicture;	12
3.1.2.13 Property LangResources: TStringList;.....	12
3.1.2.14 Property LangFileEncoding: TLFEncoding;.....	12
3.1.3 TLangManEngine Events.....	13
3.1.3.1 OnChangeLangQuery: TContinueQuery;	13
3.1.3.2 OnChangeLanguage: TNotifyEvent;	13
3.1.3.3 OnBeforeEdit: TNotifyEvent;	13
3.1.3.4 OnAfterEdit: TNotifyEvent;	13
3.2 TLangManComponent = class (TComponent).....	14
3.2.1 TLangManComponent Properties.....	14
3.2.1.1 Property LangManEngine: TLangManEngine;.....	14
3.2.2 TLangManComponent Events.....	14
3.2.2.1 OnChangeLanguage: TNotifyEvent;.....	14
3.3 TLangManClient = class (TLangManComponent).....	15
3.3.1 TLangManClient Methods.....	15
3.3.1.1 Function AddComponent (Component: TComponent; Name: string; Translate: boolean): Boolean;.....	15
3.3.1.2 Procedure RecreateTransStruct;.....	15
3.3.1.3 Procedure TranslateComponent(Component: TComponent; Name: string = ");...16	16

3.3.1.4 Procedure Translate;.....	16
3.3.2 TLangManClient Properties.....	16
3.3.2.1 Property InitAfterCreateForm: boolean;.....	16
3.3.2.2 Property TransStringProp: TTranslateStringProperties;.....	16
3.3.2.3 Property TransTStringsProp: TTranslateTStringsProperties;.....	16
3.3.2.4 Property TransStructuredProp: TTranslateStructuredProperties;	16
3.3.2.5 Property TransOtherProp: TTranslateOtherProperties;	16
3.3.2.6 Property TransAdditions: TAdditionSet;.....	17
3.4 TLexicon = class (TLangManComponent).....	17
3.5 TDesignedLexicon = class (TLexicon).....	17
3.5.1 TDesignedLexicon Methods.....	17
3.5.1.1 Function CreateItem(Text: string): Integer;.....	17
3.5.1.2 Function CompleteString(const Str: string): string;.....	17
3.5.2 TDesignedLexicon Properties.....	18
3.5.2.1 Property Item [Index: Integer]: string; (read-only).....	18
3.5.2.2 Property Items: TStringList;.....	18
3.5.2.3 Property Link [Index: Integer]: string; (read-only).....	18
3.6 TProgrammableLexicon = class (TLexicon).....	19
3.6.1 TprogrammableLexicon Methods.....	19
3.6.1.1 Procedure DefineItem(ItemNr: Word; Text: string);.....	19
3.6.1.2 Function IsDefined(Index: Integer): Boolean;.....	19
3.6.1.3 Function CompleteString(const Str: string): string;.....	19
3.6.2 TProgrammableLexicon Properties.....	19
3.6.2.1 Property Item [Index: Integer]: string; (read-only).....	19
3.6.2.2 Property Link [Index: Integer]: string; (read-only).....	19
3.6.3 TprogrammableLexicon Events.....	19
3.6.3.1 OnInitialization: TNotifyEvent;.....	19
3.7 TShadowComboBox = class (TCustomComboBox);.....	21
3.8 TLangCombo = class (TShadowComboBox).....	21
3.8.1 TLangCombo Properties.....	21
3.8.1.1 Property LangManEngine: TLangManEngine;.....	21
3.8.1.2 Property StyleCombo: TLangComboStyle;.....	21
3.8.2 TLangCombo Events.....	21
3.8.2.1 OnChangeLanguage: TNotifyEvent;.....	21
3.9 TShadowComboBoxEx = class (TCustomComboBoxEx);.....	22
3.10 TLangFlagsCombo = class (TShadowComboBoxEx).....	22
3.10.1 TLangFlagsCombo Properties.....	22
3.10.1.1 Property LangManEngine: TLangManEngine;.....	22
3.10.2 TLangFlagsCombo Events.....	22
3.10.2.1 OnChangeLanguage: TNotifyEvent;.....	22
3.11 TValuedLabel = class (TCustomLabel).....	23
3.11.1 TValuedLabel Properties.....	23
3.11.1.1 Property Value: TCaption;.....	23
3.11.1.2 Property ValueName: TCaption;.....	23
3.11.1.3 Property ValueSeparator : string;.....	23
3.11.1.4 Property ValueSpaces : byte;.....	23
3.12 TLangManStrings = class (TStringList).....	24
3.12.1 TLangManStrings Constructor.....	24
3.12.1.1 Constructor Create(ControlledStrings: TStrings; Lexicon: TLexicon);.....	24
3.12.2 TLangManStrings Methods.....	24
3.12.2.1 Procedure Translate;.....	24
4 Language files as application resources	25
5 Dynamic generating of texts	26

1 General Information

LangMan component package is used for a very easy creating of multilingual applications in Delphi. These components virtually relieve you from all the troubles with programming of translations, switching languages, and it maintains the full visibility of the source code. Unlike other competing solutions **LangMan** is a totally unique and invaluable tool.

You will create the application in any original language, regardless of the need for future translation into other languages. Only in case of strings assigned at runtime of the program, it is necessary to refer to the lexicons that are a part of this component series. When designing forms it is also useful to attend to variable length of strings. Everything else is fully automatic, and therefore **LangMan** saves a great deal of time when creating multilingual applications.

The final language files can be distributed either simultaneously with the application, but also separately. In this way languages can be easily repaired or added directly into the user's target application. Another advantage is the ability of **LangMan** to inherit languages, without any depth restriction. One language can be inherited from another language, and that one from another language and so on. It is actually highly recommended. The advantage is that in case of a missing translation of some component, the Slovak version may stem from the Czech one, or vice versa, the American from the British one, the Austrian from the German one, etc. It also allows to translate some languages only partially which, again, saves time.

LangMan itself will automatically create functional links in the selected language selection menu. For the user language selection, visual component **TLangCombo** or **TlangFlagsCombo** may be inserted into the application. All this without any strain or difficult setting. A graphic symbol or a flag can be assigned to particular languages. It will be automatically displayed in the language selection menu.

LangMan supports UNICODE charset, and it is able to translate into any language **all the standard Delphi components**. This also applies to properties inherited from the standard Delphi components, ie. if the developer applies for his component a standard Delphi constituent, also inherited properties of his component will be automatically translated. External or internal components that do not stem from standard components may be translated manually in the **OnChangeLanguage** event by using lexicons, or an automatic translation of any other components may be easily additionally programmed in the **LMAdditions** unit.

If a translation of some properties is not desirable, there are naturally several ways to disable the translation. First, I would premise, **LangMan** is intelligent enough and does not include anything useless into the translation. It translates only those properties of components, whose translation is in 99% desirable. Strings not consisting of text are not included into the translation. Strings identical with the name of the given component are not translated either. Moreover, the translation of specific properties can be easily disabled in the settings. Each form can have different settings.

1.1 Supported versions of Delphi

- Delphi 7
- Delphi 2005
- Delphi 2006
- Delphi 2007
- Delphi 2009
- Delphi 2010
- Delphi XE
- Delphi XE2

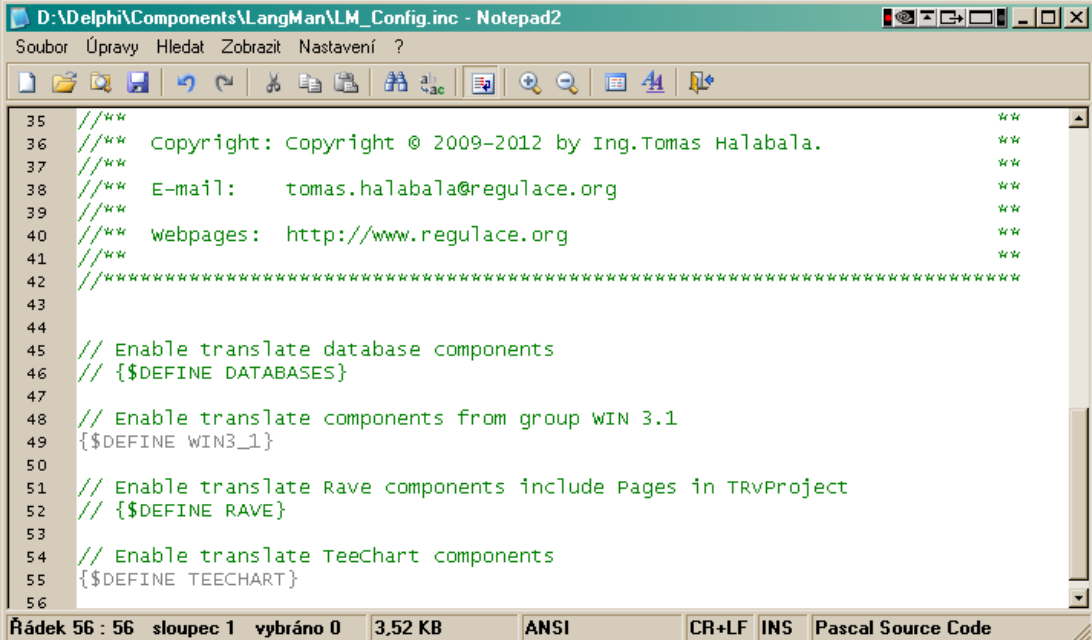
1.2 Install LangMan components

After downloading the ZIP package from the Internet you first create a folder where you extract the source code of the **LangMan** components.

So I have this folder created for example in the following location:

D:\Delphi\Components\LangMan

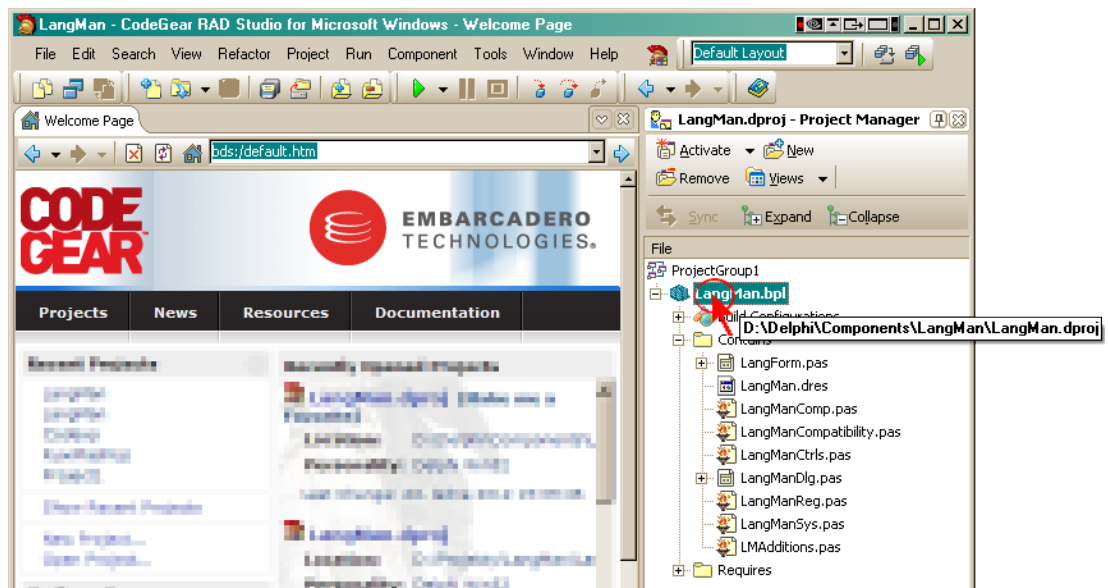
Next, open the file **LM_Config.inc** and clear definitions of the names of libraries that your edition of Delphi does not. For example, if your Delphi does not include **Rave Reports** and **database components**, you perform the following changes:



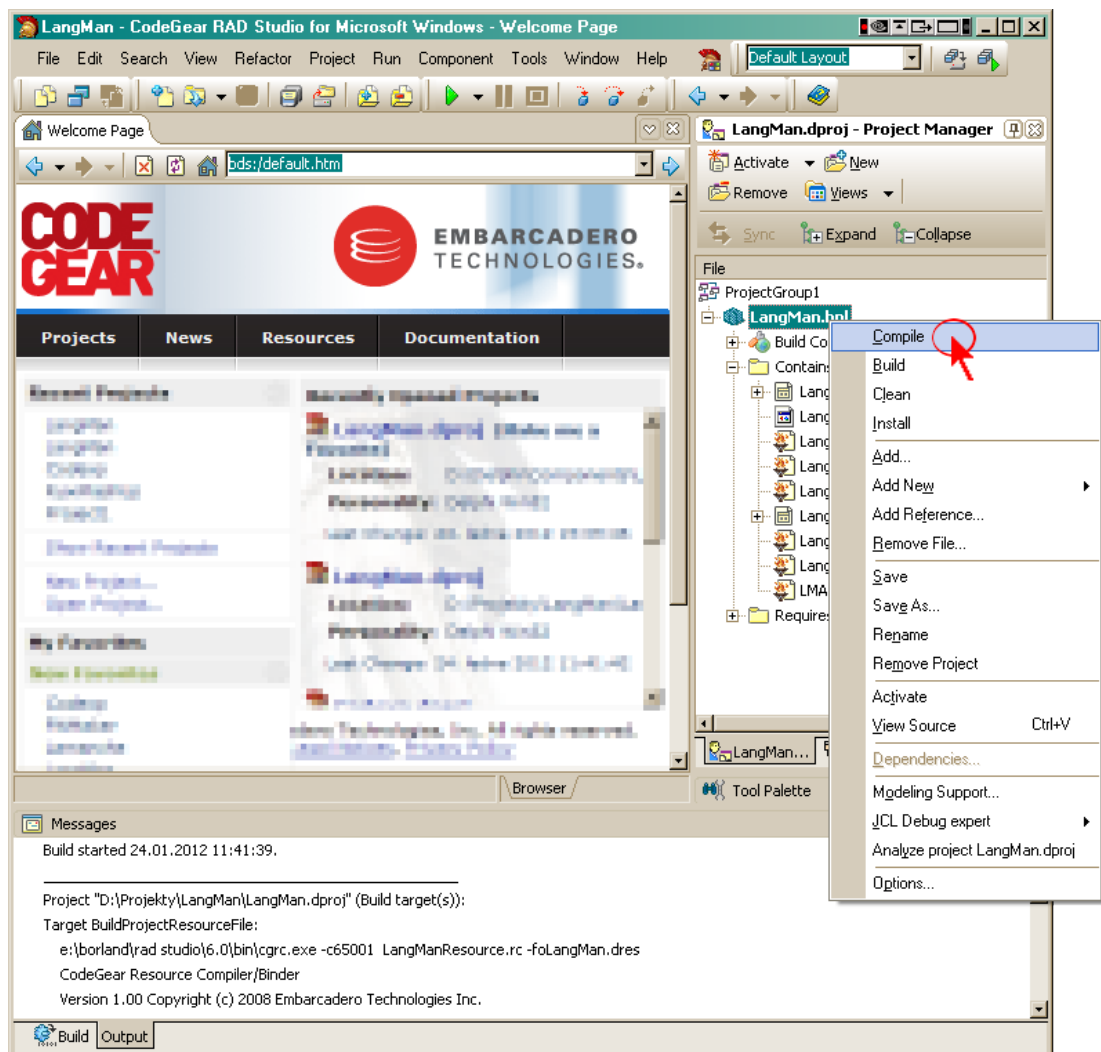
```
35 // **
36 // ** Copyright: Copyright © 2009–2012 by Ing.Tomas Halabala. **
37 // **
38 // ** E-mail: tomas.halabala@regulace.org **
39 // **
40 // ** webpages: http://www.regulace.org **
41 // **
42 // ****
43
44
45 // Enable translate database components
46 // {$DEFINE DATABASES}
47
48 // Enable translate components from group WIN 3.1
49 {$DEFINE WIN3_1}
50
51 // Enable translate Rave components include Pages in TRVProject
52 // {$DEFINE RAVE}
53
54 // Enable translate TeeChart components
55 {$DEFINE TEECHART}
56
```

Next, run your **Delphi IDE**. Then in the **File** menu, click **Open Project**. In the dialog that is displayed, select and open the file **LangMan.dpk**.

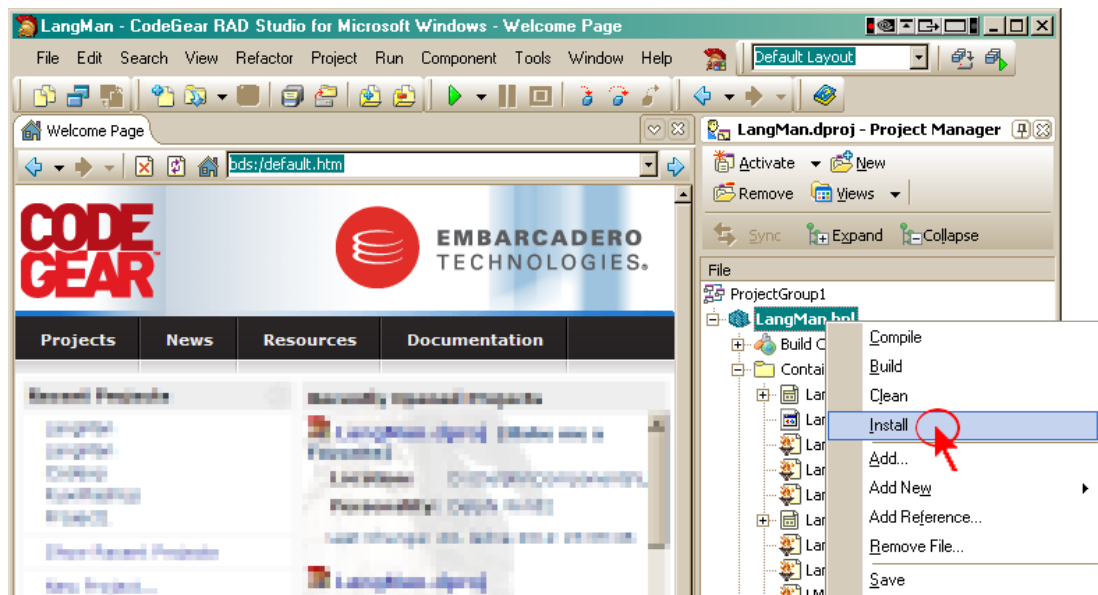
Then right click on the project name in **Project Manager**:



In the context menu, click **Compile**:



and finally in the same menu, click **Install**:



2 LangMan 1.2 Package Components

2.1 *TLangManEngine*

The basic building block, further referred to as the engine. It provides translation, switching between languages, and it manages the language files. This component is necessary for functioning of all the other language components, except the **TValuedLabel** component. In most cases, one language is common to all the parts of the program, so it is necessary to place the engine once into the unit (on the form or, rather into the **TDataModule** data module), 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 to the distinction. These characteristics determine the basic parameters of language files that will belong to given language engine.

2.2 *TLangManClient*

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.

2.3 *TDesignedLexicon*

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 entry is always read in the appropriate language. Strings in **TDesignedLexicon** are defined by using the editor directly in object inspector.

2.4 *TProgrammableLexicon*

Programmable lexicon entries can be defined, unlike **TDesignedLexicon**, 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 runtime. 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.

Another function of both lexicon 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**. By means of **CompleteString** method it is possible to transfer the text with links into the currently selected language at any time.

2.5 *TLangCombo*

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.

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**.

2.7 *TValuedLabel*

TValuedLabel is an additional component similar to the standard **TLabel**. It has **ValueName**, **ValueSeparator**, **ValueSpaces** and **Value** properties, instead of the **Caption** 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** only the **ValueName** property. It is sufficient to enter only the “**Value**” value in the program. The usage is then clear from this description.

3 LangMan Component Package Classes

3.1 *TLangManEngine = class (TComponent)*

Unit: LangManComp;

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

3.1.1 *TLangManEngine Methods*

3.1.1.1 *Function Translate (LangName: TLanguage): TLanguage;*

This function translates all assigned components into the **LangName** language. The transferred parameter is of the type of **string** and it must correspond to the name of one of the loaded languages. If the translation is realized well, an event **OnChaneLanguage** is called. After the translation, the function returns the name of the current language.

3.1.1.2 *Function GetLanguagesList: TStrings;*

The function returns the list of loaded languages. Items on this list are the valid languages and can be used as a parameter of function **Translate**.

3.1.1.3 *Function GetLangFilesList: TStrings;*

The function returns the list of language files that are loaded in the engine. List item indexes correspond to the **GetLanguagesList** indexes.

3.1.1.4 *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 default language of the design does not count.

Note: Language editor is available directly in the language menu (Menu, Combo), if it is enabled by setting of property **LangEditorVisible** to **true**.

3.1.1.5 *Procedure ShowLangCreator;*

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

Note: A new language can be added directly from the language menu (Menu, Combo), if it is enabled by setting of property **LangCreatorVisible** to **true**.

3.1.2 TLangManEngine Properties

3.1.2.1 *Property CurrentLanguage: String; (read-only)*

CurrentLanguage property returns the name of the currently selected language. This property is read-only. Language can be changed using the function **Translate**.

3.1.2.2 *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.

3.1.2.3 *Property DefaultLanguage: TLanguage;*

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.

3.1.2.4 *Property LangSubdirectory: String;*

The name of a subdirectory for storing the language files. As a default directory is considered the directory, in which there is located the application executable file. If this property is equal to an empty string, the language files will be stored in the same directory.

3.1.2.5 *Property LangFileExtension: String;*

The extension of language files. In case of a multiple usage of **TLangManEngine** 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.

3.1.2.6 *Property LangFileSignature: String;*

The identification string of language files belonging to this engine. In case of a multiple usage of **TLangManEngine** 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.

3.1.2.7 *Property LangCreatorVisible: boolean;*

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

3.1.2.8 *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 the program only includes the main language of the design and no other.

3.1.2.9 *Property TranslateLangMan: boolean;*

TranslateLangMan property allows the inclusion of the language editor into the list of translated components. If the user is not supposed to have the right to use the language editor, it means in case that the **LangEditorVisible** and **LangCreatorVisible** properties are set to **false**, there is no reason of the language editor translating. In this case, keep the property value set to **false**. The language editor currently includes three internal languages into which it can translate itself automatically. These are the Czech, the Slovak and the English language.

3.1.2.10 *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 or submenu.

3.1.2.11 *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.

3.1.2.12 *Property DesignLangFlag: TPicture;*

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

3.1.2.13 *Property LangResources: TStringList;*

To this property source language file ID names (Resources ID), linked to .exe files of the application, can be assigned in object inspector. It is therefore possible by means of this property to install the selected languages firmly into the application. If the user is allowed to edit the languages, the language file from resources will be saved to disk before editing.

3.1.2.14 *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);

3.1.3 TLangManEngine Events

3.1.3.1 *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.

3.1.3.2 *OnChangeLanguage: TNotifyEvent;*

The event called after the language change.

3.1.3.3 *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.

3.1.3.4 *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.

3.2 *TLangManComponent = class (TComponent)*

Unit: LangManComp;

TLangManComponent class is the basis for all other language components, such as **TLangManClient**, **TDesignedLexicon** and **TProgrammableLexicon**.

3.2.1 *TLangManComponent Properties*

3.2.1.1 *Property LangManEngine: TLangManEngine;*

To the **LangManEngine** property **TlangManEngine** must be assigned. It provides the current language data to the language component.

3.2.2 *TLangManComponent Events*

3.2.2.1 *OnChangeLanguage: TNotifyEvent;*

The event called after the language change.

3.3 *TLangManClient = class (TLangManComponent)*

Unit: LangManComp;

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.

3.3.1 *TLangManClient Methods*

3.3.1.1 *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 **TLangManClient** 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.

The transfer parameter **Component** must be the added component, in the parameter **Name** it is possible to transfer 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.

3.3.1.2 *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.

3.3.1.3 *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 transferred 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 transfer a new name in the parameter **Name**. Otherwise, leave an empty string to the parameter **Name**.

3.3.1.4 *Procedure Translate;*

It translates the form and all its named components. When changing the engine 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.

3.3.2 TLangManClient Properties

3.3.2.1 *Property InitAfterCreateForm: boolean;*

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

3.3.2.2 *Property TransStringProp: TTranslateStringProperties;*

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

3.3.2.3 *Property TransTStringsProp: TTranslateTStringsProperties;*

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

3.3.2.4 *Property TransStructuredProp: TTranslateStructuredProperties;*

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

3.3.2.5 *Property TransOtherProp: TTranslateOtherProperties;*

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.

3.3.2.6 *Property TransAdditions: TAdditionSet;*

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

3.4 *TLexicon = class (TLangManComponent)*

Unit: LangManComp;

The basic class of all lexicons. It provides the linkage of the lexicon with the engine. Since the 1.1 version **TLexicon** has defined virtual functions for the translation of texts containing links to the strings of the lexicon. These links are generated by another virtual function called **GetLink**. More about the function of the translation of texts and links in the description of specific lexicons.

3.5 *TDesignedLexicon = class (TLexicon)*

Unit: LangManComp;

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 **TDesignedLexicon** are defined by means of the editor directly in object inspector.

3.5.1 *TDesignedLexicon Methods*

3.5.1.1 *Function CreateItem(Text: string): Integer;*

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

3.5.1.2 *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**. This property 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 new 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.

3.5.2 TDesignedLexicon Properties

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

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

3.5.2.2 *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**.

3.5.2.3 *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.

3.6 *TProgrammableLexicon = class (TLexicon)*

Unit: LangManComp;

Programmable lexicon items can be defined, unlike **TDesignedLexicon**, 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 runtime. 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.

3.6.1 *TprogrammableLexicon Methods*

3.6.1.1 *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.

3.6.1.2 *Function IsDefined(Index: Integer): Boolean;;*

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

3.6.1.3 *Function CompleteString(const Str: string): string;*

The importance of this function is the same as in the case of both lexicons. See the description of **CompleteString** function of the **TDesignedLexicon** lexicon.

3.6.2 *TProgrammableLexicon Properties*

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

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

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

The same as in case of property **Link** of the **TDesignedLexicon** lexicon.

3.6.3 *TprogrammableLexicon Events*

3.6.3.1 *OnInitialization: TNotifyEvent;*

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

3.7 *TShadowComboBox = class (TCustomComboBox);*

Unit: LangManSys;

The declaration of shadow class for **TCustomComboBox**. The significance lies only in differentiation the **TShadowComboBox** descendants from the **TCustomComboBox** descendants on account of the automatic translation by means of **LangManClient** component. The **Items** items, classes based on **TshadowComboBox**, are not translated by **LangMan**.

3.8 *TLangCombo = class (TShadowComboBox)*

Unit: LangManComp;

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.

3.8.1 *TLangCombo Properties*

3.8.1.1 *Property LangManEngine: TLangManEngine;*

To this property **TLangManEngine** must be assigned. **TLangCombo** is linked with it.

3.8.1.2 *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.

3.8.2 *TLangCombo Events*

3.8.2.1 *OnChangeLanguage: TNotifyEvent;*

The event called after the language change.

3.9 *TShadowComboBoxEx = class (TCustomComboBoxEx);*

Unit: LangManSys;

The declaration of shadow class for **TCustomComboBoxEx**. The significance is the same as in case of **TShadowComboBox**. It serves to differentiate **TShadowComboBoxEx** descendants from **TCustomComboBoxEx** descendants, on account of the automatic translation by means of **LangManClient** component. The **Items** items, classes based on **TShadowComboBoxEx**, are not translated by **LangMan**.

3.10 *TLangFlagsCombo = class (TShadowComboBoxEx)*

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**.

3.10.1 *TLangFlagsCombo Properties*

3.10.1.1 *Property LangManEngine: TLangManEngine;*

To this property **TLangManEngine** must be assigned. **TLangFlagsCombo** is linked to it.

3.10.2 *TLangFlagsCombo Events*

3.10.2.1 *OnChangeLanguage: TNotifyEvent;*

An event called after the language change.

3.11 *TValuedLabel = class (TCustomLabel)*

Unit: LangManCtrls;

TValuedLabel is an additional component similar to the standard **TLabel**. It has **ValueName**, **ValueSeparator**, **ValueSpaces** and **Value** properties instead of the **Caption** property. These properties are linked in the the final display, while **ValueSpaces** indicates the number of spaces instead of **ValueSeparator**. The trick is that **LangMan** translates except **Hint** only the **ValueName** property. It is sufficient to enter only the **Value** value in the program.

3.11.1 *TValuedLabel Properties*

3.11.1.1 *Property Value: TCaption;*

Any string can be assigned to this property. The resulting string that is displayed by the visual component has the following format: '**ValueName**' + '**ValueSeparator**' + **ValueSpaces** x ' ' + '**Value**'. Only **ValueName** property is translated automatically.

3.11.1.2 *Property ValueName: TCaption;*

Into the translations only the property is included to which the name of the Value value in the designing language should be assigned. The resulting string that is displayed by this visual component has the following format: '**ValueName**' + '**ValueSeparator**' + **ValueSpaces** x ' ' + '**Value**'.

3.11.1.3 *Property ValueSeparator : string;*

Any string can be assigned to this property. The resulting string that is displayed by the visual component has the following format: '**ValueName**' + '**ValueSeparator**' + **ValueSpaces** x ' ' + '**Value**'. Only **ValueName** property is translated automatically.

3.11.1.4 *Property ValueSpaces : byte;*

This property indicates the number of spaces between **ValueSeparator** and **Value**. The resulting string that is displayed by the visual component has the following format: '**ValueName**' + '**ValueSeparator**' + **ValueSpaces** x ' ' + '**Value**'. Only **ValueName** property is translated automatically.

3.12 *TLangManStrings = class (TStringList)*

Unit: LangManComp;

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.

3.12.1 *TLangManStrings Constructor*

3.12.1.1 *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.

3.12.2 *TLangManStrings Methods*

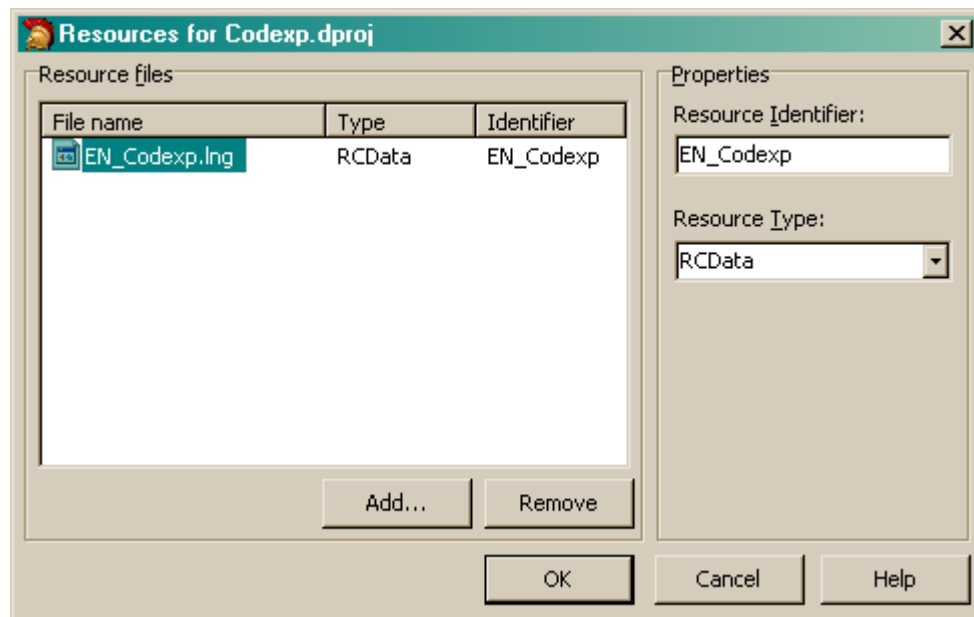
3.12.2.1 *Procedure Translate;*

This procedure will translate the entire text in the managed object, derived from **TStrings**, which was when creating the **TLangManStrings** object by means of **Create** constructor transferred in the **ControlledStrings** parameter. This procedure does not start automatically when changing the lexicon language. You must call this procedure in the right place of the program yourself. In most cases it is sufficient to insert calling of this procedure into the subprogram of the **OnChangeLanguage** event.

4 Language files as application resources

Since the 1.1 version **LangMan** components language files can be a part of an executable EXE file of an application. Below I will describe how to do it in Delphi 2009.

In the menu **Project / Resources...** open a dialog box for editing the resources, that are supposed to be independently of the program of the application linked to the resulting EXE file.



Obr. 4.1: 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 from the EXE file to the disk under the same name and with a extension assigned according to **LangFileExtension** property of the relevant 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 EXE 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 built-in language editor **LangMan** at any time and you can rely on the fact that during the following compilation every change will be introduced into the new EXE file.

Now it is still necessary to order to the relevant language engine (**TLangManEngine**) 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.

5 Dynamic generating of texts

If your program generates a statement, a log file, etc. at runtime, 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 runtime of the program, and it is even desirable to change the language of the the entire statement from the beginning not only for the following part following.

For these cases, a new class **TLangManStrings** and visual component **TLangManRichEdit** has been created and into the lexicons the method **CompleteString** and **Link** property has been added.

Example:

If you have in your application an inserted visual component **TMemo**, into which the program generates some statement at runtime, 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 (**TDesignedLexicon** or **TProgrammableLexicon**), which contains all the necessary language strings for the dynamic generation of your statement.

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 transferred 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 relevant engine, the inserted text would stay unchanged and in Czech.

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

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

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 transferred 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.

Before closing the program, remember to release the memory assigned to the **Memo** object by means of the method **Free**:

```
Memo.Free;
```

Document Revision

Datum Date	Revize Revision	Poznámka Note
11.2.2012	05	LangMan 1.2.1 Version Release
24.1.2012	04	LangMan 1.2.0 Version Release
15.4.2011	03	LangMan 1.1.8 Version Release
15.8.2010	02	LangMan 1.1.1 Version Release
17.2.2010	01	LangMan 1.1.0 Version Release

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>