1 Porting Delphi Components to Lazarus

Introduction

Porting applications to a different development environment is not an easy task. For standard, vanilla applications, this should be a relatively painless operations. But when porting Delphi applications to Lazarus, one sometimes is faced with the task to port used third-party components to Lazarus.

For most non-visual components this task should also not be very difficult: the Free Pascal compiler is meanwhile very Delphi compatible. Only when porting to new platforms, the Windows API must be replaced with something cross-platform: This is the main obstacle.

However, many existing Delphi visual components are deeply rooted in Windows and are next to impossible to port to Lazarus - unless they are rewritten from the ground up. The Lazarus structure aims to re-implement the VCL as closely as possible, but it is nevertheless sufficiently different to make porting some components a difficult task - but not impossible: simple descendents of standard Delphi components will be easily ported.

Compiler issues

The Free Pascal compiler is a bit different from Delphi: It knows multiple compiler modes, and one point to be considered when converting components is which compiler mode will be used:

- Plain FPC. This does not allow use of classes or threads or many other delphi features, so it is most likely out of the question
- TP compatibility mode has the same drawbacks as the FPC mode, so is equally useless.
- OBJFPC mode: This is the object-pascal mode of the compiler: Support for classes, exceptions, threads are switched on.
- Delphi mode: The compiler tries to be as Delphi-compatible as possible.

Obviously, the choice is between the latter two. The OBJFPC mode is more strict than the Delphi compatibility mode: FPC imposes more stringent rules on code than Delphi does. A common example is the fact that an argument to a method may not appear in the list of properties. Thus, the following code is legal in Delphi, but illegal in FPC:

```
TMyComponent = Class(TComponent)
  Procedure SetOtherValue(AValue : Integer);
  Property AValue : Integer Read FValue Write FValue;
end;
```

When referring to `AValue` in the implementation of `SetOtherValue`, it is ambiguous which instance is meant: Compiling this in FPC mode will therefore result in a duplicate identifier error. In Delphi mode (and in Delphi itself) this compiles fine.

The Lazarus team uses the OBJFPC mode, and asks that for contributions to the LCL, this mode is used.

To set the mode in a unit, just insert the following code before the line with the `Unit` keyword:

```
{$IFDEF FPC}
{$MODE DELPHI}
{$ENDIF}
```

The `IFDEF` conditional is used to make sure that Delphi itself can still compile the code, if this should be required. In case OBJFPC mode is used, the conditionals can be left out, since Delphi won't compile some `OBJFPC` compliant code.

A complete list of differences between the various compiler modes can be found in the user's manual of FPC.

The Free Pascal compiler does not support dynamic packages at this point, so if the component is implemented in a package, this package must be converted to a Lazarus package. This is an easy process: create a new lazarus package, and simply add all units in the Delphi package to the Lazars package.

2 Lazarus architecture: The LCL

A port from existing components to Lazarus, must be seen not simply as a port from one compiler (Delphi) to another (FPC), but as a port from one GUI platform (Most likely Windows) to another: the LCL (Lazarus Class Library).

Why is this so ? The LCL is not simply a rewriting of Delphi's VCL: It is a cross-platform GUI toolkit which is independent of a particular OS or particular GUI system: it provides a definite API which is the same for all platforms. A component written for the LCL will work on all platforms.

Porting to the LCL means therefore to change the Windows API mechanisms to mechanisms provided by the LCL. The LCL also has a messaging system which resembles the Windows mechanism, but there the correspondence ends. Often, many Windows API calls or messages will not be available in the LCL (This will be shown later in this article).

As shown in figure 1 on page 3 an application written using Lazarus consists essentially of 3 parts:

- Application code - the actual application.
- LCL classes: A VCL like class library.
- The LCL interface: An abstraction layer for the underlying GUI

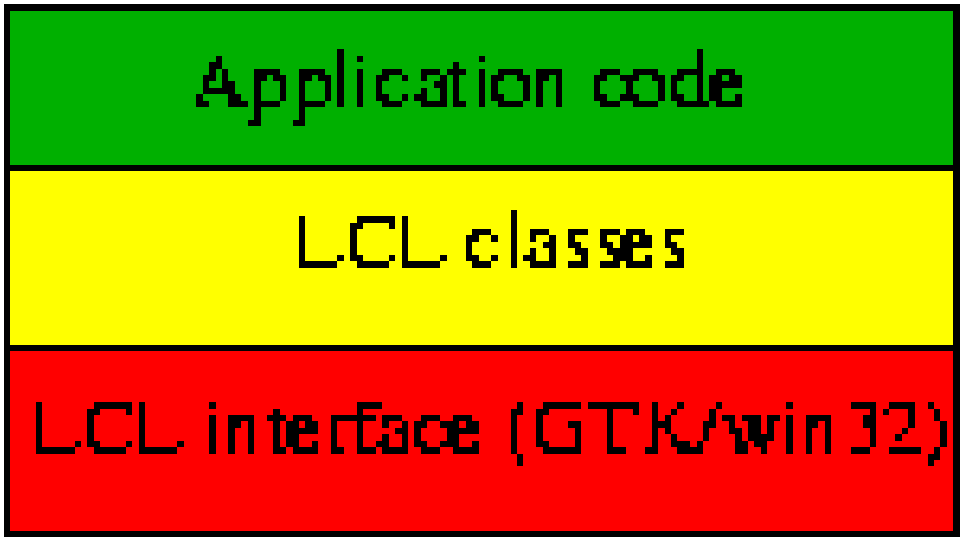
The application programmer need not concern himself with the bottom layer, the LCL interface: It should suffice to know the LCL classes - which are designed to resemble their VCL counterparts as much as possible. Someone wishing to port a component, however, must know how the abstraction layer works, since the Windows API calls must be translated to the LCL interface. In the next sections, the interface will be examined a little closer.

3 The LCL interface layer

The LCL interface layer forms a abstasction layer between the class library and the underlying GUI system: currently abstraction layers exist for GTK, GTK 2 and native Windows 32.

The layer is implemented as a single class which descends from `TInterfaceBase` (in unit interfacebase): When compiling the LCL, one descendent of this class must be instantiated - this instance will be called by all classes in the LCL. It contains all calls that

Figure 1: A lazarus application



are needed by the LCL to handle GUI events, create new windows etc: in short all calls needed to interact with the graphical system. Below is an excerpt of its interface:

```
procedure HandleEvents;
procedure WaitMessage;
procedure AppInit;
procedure AppTerminate;
function IntSendMessage3(LM_Message: Integer;
                         Sender: TObject;
                         Data: pointer): integer;
function CreateDIBitmap(DC: HDC;
                        var InfoHeader: TBitmapInfoHeader;
                        dwUsage: DWORD;
                        InitBits: PChar;
                        var InitInfo: TBitmapInfo;
                        wUsage: UINT): HBITMAP;
```

All these methods are abstract and must be implemented in a descendent. There are of course many more calls than presented here, but a complete description of all possible calls is outside the scope of this article (indeed, many issues of Toolbox could be devoted to describing the whole interface).

The interface object communicates with the classes mainly by means of messages, sometimes by means of direct method calls. The system is very much like the Windows messaging system, only the message numbers will differ: The message names have been kept similar. For example the Windows `WM_SIZE` has been renamed to `LM_SIZE` to be more platform independent.

FPC supports the concept of 'message handling' methods as introduced in Delphi, this means that for a control to respond to the `LM_SIZE` message, the following method can be implemented:

```
procedure WMSize(var Message: TLMSize); message LM_SIZE;
```

As can be seen, this is very similar to the method as it would be implemented under Delphi.

The TWMSize has been changed to TLMSize and the message identifier has been changed to LM_SIZE. The LCL layer will see to it that the message is delivered to the control.

On the other hand, the LCL classes use mainly methods of the TInterfaceBase class to talk to the interface: messages will be seldom sent from the LCL classes to the interface layer - If a component relies heavily on sending messages, chances are that it will be hard to port to the LCL.

As the messages are more considered to be an internal communication system, it is not a good idea to rely on them: In general for each message, there is a method in the LCL base classes (TControl and TWinControl) which is called by the message handler. When porting, it is therefore a good idea to use the message methods as little as possible, but instead override the methods that correspond to the message. In the case of LM_SIZE, this is the DoSetBounds method.

Unfortunately, there is no comprehensive list of supported messages nor of the methods that correspond to them. In order to find out what is possible and what not, the LCL sources must be studied. The controls unit in the LCL provides a nice overview of most message handlers, and is best studied first before looking in other places.

Interface units

As much of the Windows functionality is captured in the Windows unit delivered with Delphi, this unit will be encountered in most Delphi programs. The same is true for some other low-level units such as Messages. Needless to say, these units are not very portable. They are therefore not included in the LCL classes, but are replaced by some other units.

Other than these base units, the LCL attempts to duplicate as much of Delphi's unit and class naming scheme as possible. This means that the uses clause can stay mostly the same for an average application. However, there are some notable differences as already indicated above. Below are some points of interests.

- The functionality provided by the Windows unit is divided over many units: interfacebase for most GUI-related calls. Most type and constant definitions have been moved to the LCLType unit; They are provided mostly for compatibility.
- Messages (names as well as record definitions) have been moved in the LMessages unit.
- the IntfGraphics unit contains most of what is needed for handling images (Bitmaps).

This list will not solve all 'Identifier not found' errors that may occur when porting a component to lazarus. It may well be that the particular message or structure or call does not exist in Lazarus, in which case something else must be tried or used.

4 A case study: porting a simple component

To demonstrate all the things explained in previous sections, a small component will be ported to Lazarus. Porting to Lazarus can be done in Windows: the compiler functions, and the IDE itself is becoming more usable every day. However, there are good reasons for trying a port on Linux: the Windows unit is delivered with FPC in the Win32 distribution. Thus, the compiler will happily compile all calls to the windows system, without complaining. Not so on Linux: The Windows unit is not present, and this will force the component porter to pay attention to all occurrences of the Windows unit in the sources, and remove

them (because the compiler will report an errror). A second reason for attempting a port is the immediate verification that the component works cross-platform. Last but not least, the filesystem of Linux is case sensitive: this will force the porter of components to choose the correct casing for the units (s)he wishes to port.

The component which will be ported is an edit component (TEditBtn) which is a regular edit control, with a speedbutton attached to it: the speedbutton is generally used to pop up a dialog to allow the user to select a value - the value will then be filled in in the edit control. Examples could be:

- Showing a file dialog to select a file.
- Showing a directory dialog to select a directory.
- Showing a calendar to pick a date.
- Showing a calculator to perform a calculation.
- Showing a form with a grid containing a dataset and controls to look for a values in the dataset.
- Anything else one might want to do.

In fact, specialized descendents of the general purpose component are being created to perform the first 4 tasks.

The original component TEditBtn (by Louis Louw) was found on Torry's pages. The sources can be found on the CD-Rom that comes with this issue. The class declaration is quite simple:

```
TEditBtn = class(TEdit)
  FButton: TSpeedButton;
  FEditorEnabled: Boolean;
  FOnBtnClick : TNotifyEvent;
  procedure SetGlyph(Pic: TBitmap);
  function GetGlyph : TBitmap;
  procedure SetNumGlyphs(ANumber: Integer);
  function GetNumGlyphs:Integer;
  function GetMinHeight: Integer;
  procedure SetEditRect;
  procedure WMSize(var Message: TWMSize); message WM_SIZE;
  procedure CMEnter(var Message: TCMGotFocus); message CM_ENTER;
  procedure CMExit(var Message: TCMExit);   message CM_EXIT;
  procedure WMPaste(var Message: TWMPaste);   message WM_PASTE;
  procedure WMCut(var Message: TWMCut);   message WM_CUT;
protected
  procedure GetChildren(Proc: TGetChildProc; Root: TComponent); override;
  function IsValidChar(Key: Char): Boolean; virtual;
  procedure aClick (Sender: TObject); virtual;
  procedure KeyDown(var Key: Word; Shift: TShiftState); override;
  procedure KeyPress(var Key: Char); override;
  procedure CreateParams(var Params: TCreateParams); override;
  procedure CreateWnd; override;
public
  constructor Create(AOwner: TComponent); override;
  destructor Destroy; override;
  property Button: TSpeedButton read FButton;
end;
```

The published properties have been left out. From the listing it is clear what methods will need to be looked at: the message methods to start with. The `CreateParams` method and `CreateWnd` calls are also dangerous: on Windows they serve to create the windows handle for the component. As the LCL has it's own mechanisms for creating a windows handle, these calls also will need to be looked at.

But first the uses clause will be taken care of. The original uses clause reads as follows:

```
uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls,
  Forms, Dialogs, StdCtrls, Mask, Buttons,ExtCtrls;
```

Keeping in mind the earlier remarks, this is changed to:

```
uses
  LCLType,LMessages, Messages, SysUtils, Classes, Graphics,
  Controls, Forms, Dialogs, StdCtrls, Buttons,ExtCtrls;
```

The 'Mask' unit has been removed, as it isn't needed at all (it's also not present in the LCL).

On the message methods the following actions are performed:

- The `WMSize` call is removed: the necessary actions will be taken in the `DoSetBounds` method which is overridden in the component.
- The `CM_ENTER` message is defined but does not work. Instead the `LM_SETFOCUS` message exists. The corresponding `TCMGotFocus` record is called `TLMSetFocus`. Taking this into account, the declaration of the `CMEnter` method is changed to the following:

  ```
  procedure WMSetFocus(var Message: TLMSetFocus); message LM_SETFOCUS;
  ```

  The `TCMGotFocus` message record is not defined: the general `TLMessage` record is substituted.
- Likewise, the message `TCMExit` does not exist, which leads to the following declaration of the `CMExit` method:

  ```
  procedure WMKillFocus(var Message: TLMKillFocus); message LM_KILLFOCUS;
  ```

- The `WMPaste` and `WMCut` methods are removed because they are not yet supported by Lazarus, although the message constants are defined: `LM_PASTEFROMCLIP` and `LM_CUTTOCLIP` are defined.
- The `Createwnd` and `CreateParams` have been removed, since the LCL doesn't provide the same functionality as needed by these methods.
- The following methods have also been removed, as they are useless and serve no function in the original component.

  ```
  function IsValidChar(Key: Char): Boolean; virtual;
  procedure KeyDown(var Key: Word; Shift: TShiftState); override;
  procedure KeyPress(var Key: Char); override;
  ```

The implementation of these methods presents no problems: They compile as-is.

The implementation of some methods also needs some work. The Constructor comes first:

```
constructor TEditBtn.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);
  FButton := TSpeedButton.Create (Self);
  FButton.Width := 15;
  FButton.Height := 17;
  IF  csDesigning in ComponentState then
   FButton.Visible := True
  Else FButton.Visible := False;
  FButton.Parent := Self;
  FButton.OnClick := aClick;
  FButton.Cursor:=crArrow;
  ControlStyle := ControlStyle - [csSetCaption];
  FEditorEnabled := True;
end;
```

At first sight, there is nothing wrong with this code. The only code that actually needs changing is the setting of the parent: In Lazarus, not any widget can contain child widgets. Therefore the line which sets the parent must be changed, so the form is made parent of the button. This also means that the top,left coordinates of the button will be measured relative to the form, rather than relative to the button. This must be taken care of in subsequent methods. All the rest of the code will compile and work just fine.

The changed constructor looks as follows:

```
inherited Create(AOwner);
FButton := TSpeedButton.Create(Self);
FButton.Width := Self.Height;
FButton.Height := Self.Height;
FButton.FreeNotification(Self);
CheckButtonVisible;
FButton.OnClick := @DoButtonClick;
FButton.Cursor := crArrow;
ControlStyle := ControlStyle - [csSetCaption];
FDirectInput := True;
```

Some names of variables have been changed for clarity. The `CheckButtonVisible` method checks whether the button should be made visible or not: The `ButtonOnlyWhenFocused` property determines whether the button is visible when the edit control does not have focus. Since the check must be made at various places, the code for it has been moved to a separate procedure. The button width can be set by the user throught the `ButtonWidth` property, and initially, the button is made square: the height is the same as the width.

The next method that causes problems is the `SetEditRect`:

```
procedure TEditBtn.SetEditRect;
var
  Loc: TRect;
begin
  SendMessage(Handle, EM_GETRECT, 0, LongInt(@Loc));
  Loc.Bottom := ClientHeight + 1;  {+1 is workaround for windows paint bug}
  Loc.Right := ClientWidth - FButton.Width - 2;
  Loc.Top := 0;
  Loc.Left := 0;
  SendMessage(Handle, EM_SETRECTNP, 0, LongInt(@Loc));
```

```
  SendMessage(Handle, EM_GETRECT, 0, LongInt(@Loc));  {debug}
end;
```

Under Windows, this call serves to limit the editing area of the edit control: The button is drawn inside the edit control. This code makes sure the cursor will never disappear 'under' the button by telling windows that only the area not under the button may be used for editing. This call disappears for 2 connected reasons:

1. The button is drawn outside the edit control in the LCL
2. The call is not supported in Lazarus: an edit control cannot contain a button, so adding the call would make little sense.

Therefore, the call is removed.

The next method implementation that needs checking is the `WMSize` message handler. Its functionality has been moved to the `DoSetBounds` call:

```
procedure TCustomEditButton.DoSetBounds(ALeft, ATop, AWidth, AHeight:
Integer);
begin
  inherited DoSetBounds(ALeft, ATop, AWidth, AHeight);
  DoPositionButton;
end;
```

The actual positioning of the button is handled in the `DoPositionButton` call (made virtual, so it can be overridden):

```
procedure TCustomEditButton.DoPositionButton;

begin
  if (FButton<>nil) then
    FButton.SetBounds(Left+Width, Top, FButton.Width, Height);
end;
```

This method does not do anything special. The button width is kept, as it can be set by the user through the 'ButtonWidth' property.

The `GetMinHeight` method is removed, as it is not really needed: The LCL makes sure the controls are big enough.

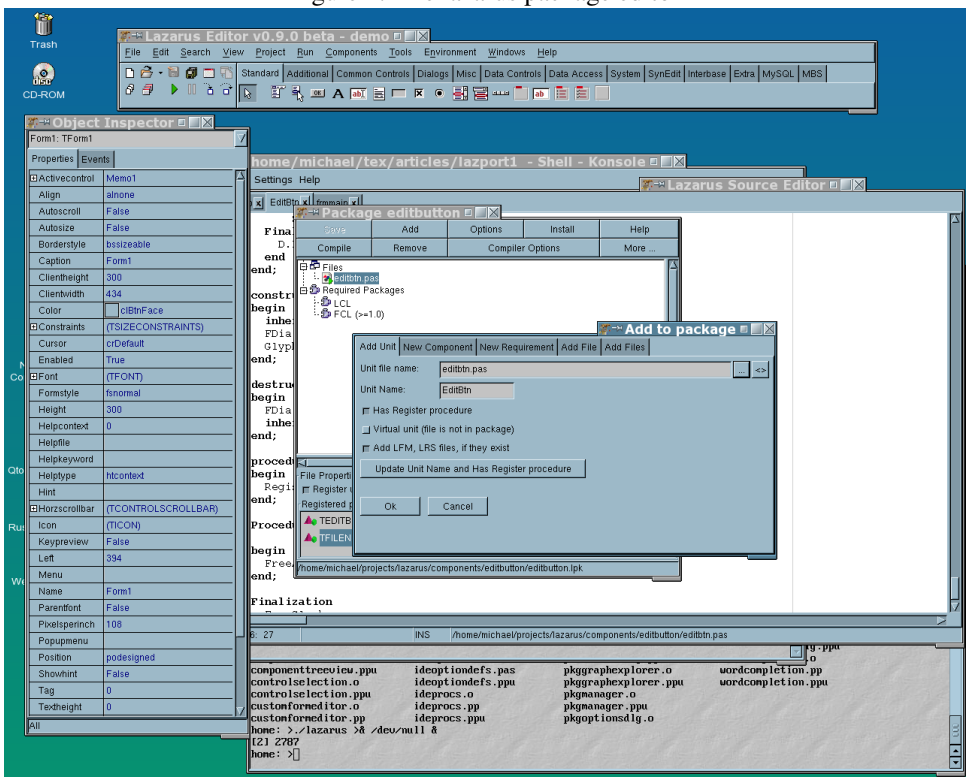
With these changes, the component should compile. The code of the final component can be found on the CD-Rom accompagnying this issue. It will most likely be included as a core component of Lazarus, together with some descendent components.

5 Installing the component in Lazarus

Now that the code compiles with Lazarus, it would be nice to have the component visible on the Component palette. In Delphi, this would be done using a package: One would create a package, insert the unit in it, recompile the package and install it in the IDE. For single components such as the editbutton, the special dclusr package can be used: it contains all components that are not explicitly part of a package.

Unfortunately, Lazarus (or better, FPC) does not yet support dynamically loading packages. But the Lazarus designers have made it really easy to install new components: Since

Figure 2: The lazarus package editor



Lazarus is open source, it comes with the sources. Thus, to install a new component, it is possible to just recompile lazarus with the new component included.

The IDE has 2 menu options for this under the `Tools` menu:

Build Lazarus this will recompile lazarus, and will add any packages marked for inclusion to the component palette.

Configure 'Build Lazarus' This allows to configure the actions taken when 'Build Lazarus' is called. In this dialog, the 'With packages' option must be set if packages with components must be included.

So, to include the edit button to the component palette, first the 'With packages' option in the configuration must be set. Then the editbtn unit must be added to an existing package, or a new package must be created. The 'File|New' menu pops up a dialog which contains a 'Standard Package' entry: this must be chosen to create a new package. The package editor will be opened: In the package editor, the 'Add' button can be used to add a unit to the package, as can be seen in figure 2 on page 9. When adding the unit, lazarus must be told whether the unit has a 'register' procedure to register components. If the flag 'Has register procedure' is set, then Lazarus will call the register procedure of the unit. As a result the component will be included on the component palette.

The 'Virtual unit' option is meant for cases when the package registers components in units that are not part of the package file list. In that case the units not contained in the package are added to the file list as 'virtual units'. This allows Lazarus to calculate package dependencies in a correct way. As an example the mysqllaz or interbaselaz packages can be viewed: They are distributed standard with Lazarus, and all contain virtual units.

Thus, to register the editbutton, the 'editbtn' unit must be added to a package (let it be

called 'editbutton'), and the 'has register procedure' flag must be set.

The package can be marked for inclusion in the IDE with the 'Install' button of the Package editor. Lazarus will add it to the list of packages marked for inclusion, and next time when the IDE is built, the package will be included: All that is left to do is install the package, rebuild lazarus from the 'Tools' menu, and after everything was recompiled succesfully, restart lazarus. There should be a new page called 'MBS' which contains the editbutton.

6 Adding an Icon: Resources

The usual way to add an icon to the component palette in Delphi is to create a resource file with a bitmap that has the name of the component. This is no different in Lazarus, only the procedure to add the resource is different, as the FPC compiler does not support including of resources on Linux.

So, to add an icon to the Lazarus component palette the following must be done:

1. Extract the bitmaps from the original delphi resources. This can be done with the image editor from Delphi. Optionally recreate them from zero.
2. Convert the image to XPM format.
3. If the lazres tool did not come in compiled form with your copy of Lazarus, compile and install it:

   ```
   cd lazarus/tools
   make all
   make install
   ```

   This should install the 'lazres' tool in your path.
4. On the command-line, convert the XPM image to a lazarus resource:

   ```
   lazres editbtn.lrs editbtn.xpm
   ```

   More than one image name can be added on the command-line, each image will be inserted in the resources.
5. Add the lresources unit to the `uses` clause of a unit (usually the unit where the component is registered).
6. Include the .lrs file in the initialization section of the unit. As an alternative, a separate procedure can be created for this, which is then called from the initialization section of the unit:

   ```
   Procedure LoadResources;

   begin
     {$i editbtn.lrs}
   end;

   Initialization
     LoadResources;
   end.
   ```

   In general, the resources must be loaded before they can be accessed from the program.

The same procedure can be followed to add about any resource to the binary. The 'Lazarus-Resources' instance class manages all resources added in this way. For example, to load an image that was added as a resource, use the following code:

```
With TPixmap.Create do
  begin
  LoadFromLazarusResource('TEditButton');
  // other code
  end;
end;
```

The resources are searched in a case-insensitive manner.

7 Conclusion

Hopefully it has been made clear that porting Delphi components to Lazarus is possible - but maybe not easy. A small case study was presented, and various issues that most certainly will show up, have been pointed out. This concludes porting compontents to Lazarus: the next step is porting complete projects, but this is left for a future contribution.

In the meantime, more information about porting components can be found on the Lazarus-CCR site on sourceforge:

`http://lazarus-ccr.sourceforge.net/`

The 'WIKI' section contains also some instructions on porting components to Lazarus. Last but not least the author wishes to thank Mattias Gaertner, one of the main Lazarus developers: without his answers and his bugfixes in Lazarus and the LCL, this article would not have been written.