

# 1 Effiziente Mehrsprachigkeit für ObjectPascal

Unter Lokalisierung oder Mehrsprachigkeit versteht man den Vorgang, bei dem man ein Programm übersetzt und auf den Einsatz in bestimmten Ländern vorbereitet. Neben der Übersetzung der Texte, die in der GUI oder einer DB vorkommen, gehört zur Lokalisierung auch die Anpassung von Programmfunktionen, wie eine Devisenbewirtschaftung, an landestypische Elemente.

Die hier vorgestellte Unit lässt sich für die Plattformen Delphi, Lazarus und Kylix einsetzen. Nachdem man das Programm und zugehörige Ressourcen für die Lokalisierung vorbereitet (internationalisiert) hat, lassen sich die Texte intern oder extern verwalten und dann entsprechend vertreiben.

## 1.1 Wirkungsweise und Einsatzgebiet

Die Übersetzungs-Tools von Borland ermöglichen auch die Lokalisierung der Software, sind aber von der IDE und zugehörigen Versionen ziemlich abhängig und entsprechend schwerfällig. Auch der externe Translation-Manager (ETM) setzt viel Verwaltungsaufwand voraus. Die einfache aber effiziente Technik einer Unit, genannt MultilangTranslator, und einer zugehörigen Ressourcendatei soll den Anforderungen für kleine und mittlere Projekte vollauf genügen.

Worauf beruht nun die Unit. Eine ausführbare Datei, die DLL's oder Packages (bpl), aus denen die Anwendung besteht, enthalten ja alle benötigten Ressourcen. Die Technik, die in Form einer einzigen Unit daherkommt, basiert auf diesen Ressourcen und dem assoziativen Speichern von Sprachverweisen in der Tag Eigenschaft (property Tag: Longint;). Ein Tag einer visuellen Komponente hat ja keine vordefinierte Bedeutung und steht demzufolge zur freien Benutzung.

Die Unit leistet folgendes:

1. Separation der Strings zur Designzeit
2. Alle Sprachen lassen sich in eine Datei linken
3. Sprachwechsel zur Laufzeit möglich
4. Keine Lizenz, Komponente oder zusätzliche Tools nötig
5. Einsatz in Delphi, Lazarus und Kylix (siehe laz\_multilang.zip)
6. Schnell und einfach zum Verteilen
7. Verständliches Ändern oder Korrigieren der Texte (Stringtable)
8. Erweiterbar durch Stringliterals oder zusätzliche Controls
9. Ereignis OnLanguageChanged() implementiert
10. Ressourcen-DLL zur Laufzeit möglich

Sicher gibt es auch Nachteile. Jede Komponente hat nur ein Tag und die Zuordnung Tag zur StringID muss man selbst verwalten. Bauen wir das Rahmenwerk nun als Beispiel Schritt für Schritt auf. Als erstes benötigen wir eine Ressourcen-Datei mit der Endung \*.rc. Die folgenden Einträge sind im Windows-Ressourcendateiformat (.RC) gespeichert:

```
/*filenameSTR.RC*/  
/*Nur ein Abschnitt für alle Sprachen*/  
STRINGTABLE  
{  
    3, "Arbeiten im Team"  
    1003, "work in team"  
    2003, "travailler en groupe"
```

```

    3003, "lavorare nel gruppo"
    4003, "trabajo en equipo"
}

```

Nach dem Erstellen und Speichern der kleinen „fünfsprachigen“ Datei wechsele ich in ein Form, z.B. der Caption des Forms, und es erfolgt die Zuweisung der Zahl 3 im Tag. Die in der Datei vorhandenen Offsets von 1000 dienen der Sprachsteuerung und garantieren auch das dynamische Wechseln der Sprache zur Laufzeit. Nach der Verknüpfung der Ressourcendatei mit der Tag Eigenschaft steht bereits der Compiler auf dem Programm. So nebenbei fügen Sie noch die Unit *MultilangTranslator.pas* in ihr Projekt ein.

Die Ressourcendatei soll jetzt kompiliert werden, entweder mit dem Ressourcen - Compiler (Brcc32.exe) oder direkt in der Projektdatei. Innerhalb der Projektdatei lässt sich mit der Compilerdirektive gleich ein binäres Ressourcenformat und das Linken derselben Datei bewerkstelligen:

```
{$R 'filenameSTR.res' 'filenameSTR.RC'}
```

Die Compiler-Direktive \$R bewirkt, dass die Datei mit rc-Endung und der kompilierten Namensweiterung \*.res in das Projekt gelinkt wird. Die res-Datei wird also aus der neuen rc-Datei kompiliert. Aber Achtung: Falls die Namen der \*.rc- und der \*.res-Datei in der Direktive \$R nicht übereinstimmen, dann verwendet das Produkt die falsche \*.res-Datei. Diesen Akt kann ich auch optional über den Kommandozeilen Compiler anstossen:

```
D:\Programme\Borland\Delphi7\Bin\brc32.exe -r filenameSTR.RC
```

Soweit sind die Vorbereitungen erledigt, nun will ich die Caption des Forms mal übersetzen lassen. Es erfolgt die eigentliche Sprachsteuerung. Im Konstruktor des Form lege ich die Sprache fest, zwei Zeilen Quellcode genügen:

```
objMultilang:= TMultiLangSC.Create(self);
objMultilang.LanguageOffset:= 1000;
```

Es sollte sich ein erster Erfolg einstellen, wie beim Holzhacken. Nach dem Starten der Anwendung wechselt das Funktionsmuster den Text auf Englisch (1003). Sie definieren in der Ressourcendatei welche Sprache welchem Offset entspricht. Die Reihenfolge Deutsch, Englisch, Französisch, Italienisch und Spanisch kann als Entwurf gelten. Auch das dynamische Übersetzen ist mit folgendem Konstrukt bereits implementiert:

```

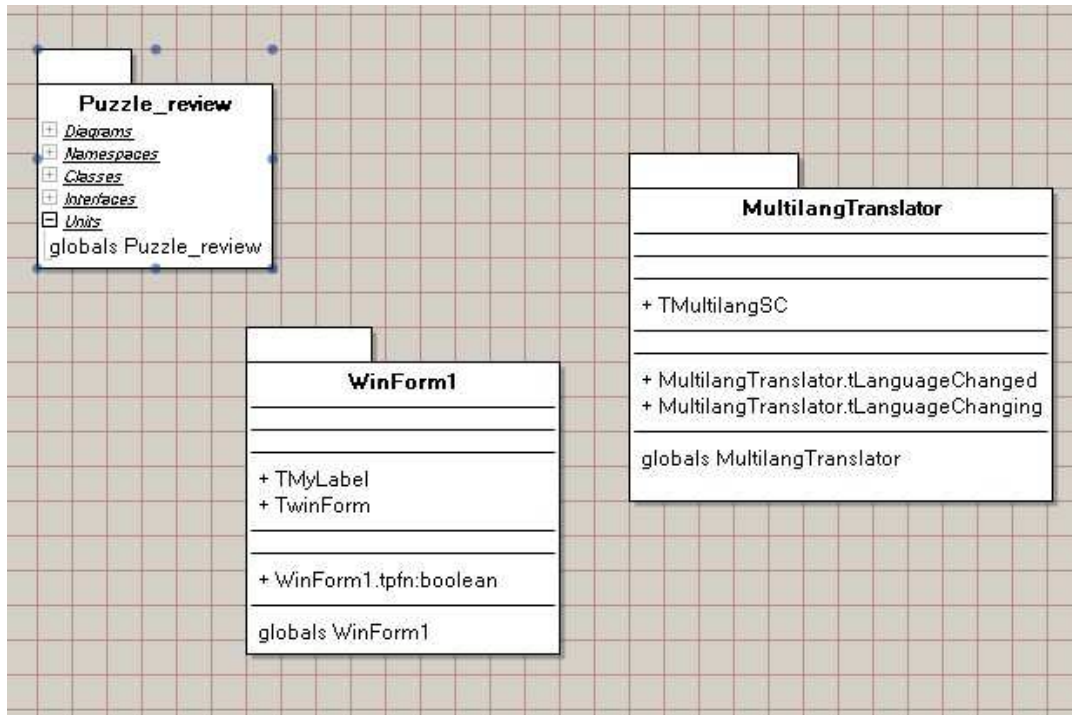
case langRGroup1.itemindex of
  0: objMultilang.LanguageOffset:= 0; //D
  1: objMultilang.LanguageOffset:= 1000; //E
  2: objMultilang.LanguageOffset:= 2000; //F
  3: objMultilang.LanguageOffset:= 3000; //I
  4: objMultilang.LanguageOffset:= 4000; //S
end;

```

Die Engine übersetzt beim Aufruf des Offset jeweils durch Iteration die Texte. Die grosse Flexibilität zeigt sich auch zur Designzeit. Die textorientierte RC-Datei lässt sich in Delphi einbinden, eine Korrektur am Text ist dann unmittelbar möglich und nach erneutem Kompilieren bereits wirksam. Vermeiden Sie aber das Aufnehmen der Datei ins Projekt, ansonsten die Ressourcen zweimal referenziert sind. Interessanterweise parst ab Delphi 2005 der Compiler auch rc-Dateien, so dass die Syntax schon im Editor einer Validierung unterliegt.

Das Framework tangiert also, wie in folgendem Packet Diagramm ersichtlich, ganze drei Units:

Die Unit *Puzzle\_review.dpr* beinhaltet die Compilerdirektive, die den Linkvorgang der Ressourcen steuert. Die Unit *WinForm1.pas* startet im Form Konstruktor den Übersetzungsvorgang, der durch die Unit/Komponente *MultilangTranslator.pas* zur Start- oder Laufzeit ermöglicht wird. Da ich die Klasse *TMultilangSC* dynamisch erzeuge, ist ein Installieren der Komponente zur Designzeit nicht nötig.



// puzzle\_multilang1.jpg

Abb.1 Das Packetdiagramm ist Mehrsprachig

Wie funktioniert nun der Ablauf, werfen wir einen Blick in die Sequenz der Engine der Klasse *TMultilangSC*. Es kann mit der Funktion *currentSystemLanguage* oder *currentLanguage* die Systemsprache oder Einstellungen innerhalb der Registry abgefragt werden (letzteres in Lazarus oder Kylix nicht nötig).

```

function TMultilangSC.currentLanguage: integer;

    property LanguageOffset: integer read fLanguage
        write SetLanguage;
        SetLanguage(const Value: integer);

    procedure TMultilangSC.ChangeLanguage(const languageOffset:
        integer);
        ChangeComponent(GetTopComponent, languageOffset);
        if Assigned(fOnLanguageChanged) then
            fOnLanguageChanged(Self);
  
```

Wer es lieber direkter mag, kann mit dem property *LanguageOffset* die Sprache so setzen, dass die Methode *SetLanguage* unmittelbar *ChangeLanguage* feuert.

Die Hauptarbeit des digitalen Dolmetschers erfolgt dann in *ChangeComponent*, welche rekursiv mit *component.count* und der tag-Steuerung die Texte einzeln übersetzt. Dazu bedienen wir uns via *getResourceString* der bekannten API-Funktion *LoadString()*:

```

begin
    if theComponent.ComponentCount > 0 then begin
        for x:= 0 to theComponent.ComponentCount-1 do
  
```

```

        ChangeComponent(theComponent.Components[x], theLanguageOffset);
end;
if theComponent.tag <> 0 then begin
    if (theComponent is TForm) then
        (theComponent as TForm).Caption:= GetResourceString(theComponent.tag)
    else if (theComponent is TLabel) then
        (theComponent as TLabel).Caption:= GetResourceString(theComponent.tag)
    else if (theComponent is TCheckBox) then .....

```

Mit der Funktion `LoadString()` lädt man einen entsprechenden String aus der Ressourcen-Skriptdatei. Der erste Parameter ist der Instanz-Handle, der zweite die ID-Kennung des Strings in der Ressource-Datei, der dritte Parameter ist der Puffer, in dem der geladene String daherkommt und mit dem vierten Parameter ist maximale Anzahl Zeichen im Puffer gefragt.

```

function TMultilangSC.GetResourceString(const number : integer) : string;
// compile with {-Sd}
var pP: array[0..255] of char;
begin
    //pP.PChar;
    if LoadString(HInstance, number+ fLanguage, pP, sizeof(pP)) > 0 then
        result:= pP
    else
        result:= '';
end;

```

Ähnlich lässt sich auch mit der Funktion `LoadIcon()`, aus der Ressourcen-Skriptdatei (\*.rc) ein Icon laden, welches mit dem Typen `ICON` angegeben wurde. Jede String-Ressource ID verweist übrigens auf bis zu 16 Strings. Dadurch wird die Kapazität für verfügbare String Quellen erhöht. `HInstance` selbst stellt ein eindeutiges Instanzen-Handle für die Anwendung oder die Bibliothek (folgt gleich) bereit, in unserem Fall die EXE mit den gelinkten Ressourcen.

Beim Gebrauch der Unit als Komponente auf den Forms kommt noch `Loaded` ins Spiel. `Loaded` ermöglicht einer Komponente, sich selbst zu initialisieren, nachdem alle ihre Teile aus einem Stream geladen wurden. Wenn das Streamingsystem ein Formular oder Datenmodul aus der entsprechenden Formulardatei lädt, erstellt es zuerst mit Hilfe des zugehörigen Konstruktors die Formalkomponente und initialisiert dann deren Eigenschaften mit den ausgelesenen Werten.

## 1.2 Pakete zur Laufzeit

Eine weitere Flexibilisierung besteht aus der Auslagerung der Ressourcen (sozusagen Ressourcen outsourcen ;) in eine dynamisch ladbare Sprachen DLL. Die String-Ressourcen werden dann nicht in die Formulardatei eingebunden, sondern in eine eigene DLL gepackt.

Das Auslagern von Ressourcen vereinfacht den Übersetzungsprozess. Eine fortgeschrittene Stufe der Auslagerung besteht im Erstellen von Ressourcenmodulen. Eine Anwendung, die Ressourcenmodule verwendet, unterstützt eine Vielzahl von Übersetzungen, indem man einfach das Ressourcenmodul austauscht.

Wie aber kommen wir an das Handle? Mit `LoadLibrary()`! Wir erhalten einen einzigartigen Namen, der nicht aus einer Zeichenfolge sondern aus einem vorzeichenlosen 16-Bit-Integer (`HANDLE`) besteht. Windows weist ihn von sich aus zu. `HInstance` wird dann von `RegisterClass` und `CreateWindow` verwendet.

Auch das Packen der Ressourcen-Texte in eine DLL geschieht erschreckend einfach. Eine DLL die nur Ressourcen enthält und diese dann zur Laufzeit benötigt, lässt sich auch ohne Experten einrichten; die Idee ist, einen Wrapper um die res-Datei zu bauen:

```

library reslang;

```

```

uses
  SysUtils;

{$R 'filename.res'}

begin
end.

```

Im laufenden Einsatz gilt es dann, die DLL zu laden um den Zugriff auf das Handle zu ermöglichen:

```

procedure TForm1.FormCreate(Sender: TObject);
const
  badDLLload = 1;
var
  h: tHandle;
  pP: array[0..255] of char;
begin
  h:= loadLibrary('reslang.dll');
  if h <= badDLLload then
    showmessage('no dll load')
  else begin
    if loadString(h, 5, pP, sizeof(pP)) > 0 then
      showmessage((pP));
    //...
  end;
end;

```

Die Methode `getResourceString` kann ich nun einfach mit dem Handle austauschen, das wiederum einzigartig ist und den Zutritt zu den Zeichenketten erlaubt.

Als Tipp lassen sich Direktiven auch unterscheiden

```

{$IFDEF MEMSOUND}
  {$R memsound.RES}
{$ELSE}
  {$R memory4.RES}
{$ENDIF MEMSOUND}

```

Von Interesse ist noch das Übersetzen von Strings direkt im Text, d.h. Text der sich nicht in den Komponenten (caption, hint) befindet. Die Lösung ist sich direkt auf die ressourcenstrings oder stringtables zu beziehen:

```

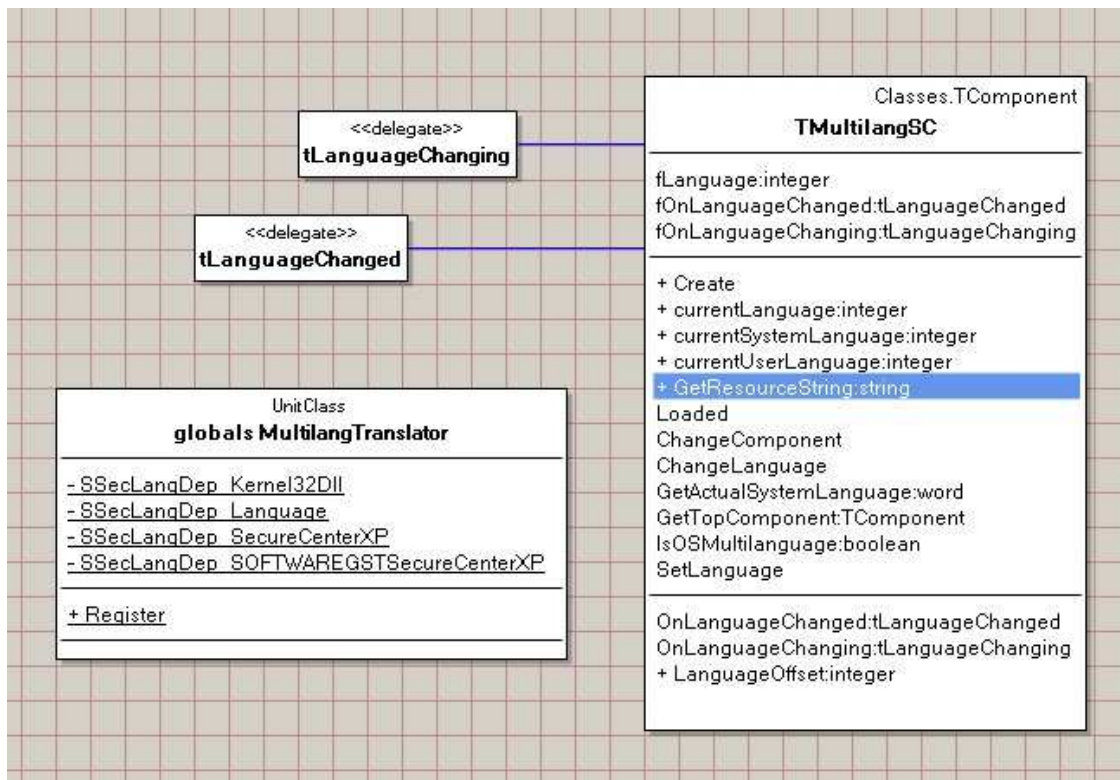
showmessage(objMultilang.GetResourceString(21));

memInfo.lines.Add(objMultilang.GetResourceString(11));
memInfo.lines.Add('');
memInfo.lines.Add(objMultilang.GetResourceString(12));

```

Beim Einsatz von Ressourcen ist man auf einige Tools angewiesen. Essentiell ist sicher der bereits bekannte Ressourcen Compiler BRCC32.EXE, der in der Version 5.4 vorliegt. Wenn der Ressourcen Compiler mit dem Client mitgeht, benötigt man zusätzlich die Datei *rw32core.dll* im Installationsumfang. Das eigentliche Bearbeiten oder Erzeugen von Ressourcen kann mit dem guten alten Resource Workshop erfolgen, der mit Delphi 5 Professional und Enterprise daherkommt oder im Borland RAD Pack enthalten ist. Der Resource Workshop 5.0 kompiliert und dekompiliert 16 und 32 Bit Ressourcen aus rc, res, exe, dll, drv, vbx, cpl, ico, bmp, rle, dlg, fnt, und cur Dateien. Diejenigen

Zeitgenossen aus der Borland Pascal Ära unter uns agierten sicher auch als Ressourcenjäger, um die tollsten Ikonen zu sammeln ;).



//multilang\_class1.jpg

Abb.2: In allen 3 Plattformen lässt sich das Laden der Ressourcen kapseln.

### 1.3 Kylix und Linux auch multilingual

Am Schluß meines Streifzuges sollen die stetig zunehmenden Linux - Entwickler auf ihre Kosten kommen. Ob schnell aber stetig auf Delphi.NET umgestiegen wird, lässt sich noch nicht beurteilen. In Kylix existiert das Tool Resbind, das aus einer EXE Ressourcen extrahieren kann. Mit dem Aufruf Resbind -r myprogram.res myprogram kommt man auch ohne Workshop zu den Ressourcen. Es ist auch möglich Win-Ressourcen in ein Shared Object (Linux DLL) mit resbind zu konvertieren. Wie aber bindet man die Ressourcen?

Eine Res-Datei für Kylix kann man mit dem Ressourcen Compiler unter Windows generieren. Kylix unterstützt aber wenig vordefinierte Typen wie ein Bitmap. Deshalb muß man alle Ressourcen, die einen vordefinierten ResType haben, als RCDATA definieren. Im Folgenden schreibe ich eine Ressource in eine Datei:

```

uses
  SysUtils, Types, Classes, Variants, QGraphics, QControls,
  QForms, QDialogs, QStdCtrls;
implementation
  {$R *.xlfm}
  {$R userdefined.res}
procedure TForm1.Button1Click(Sender: TObject);
var stm: TResourceStream;
begin
  stm:= TResourceStream.Create(HInstance, 'MYRES1', RT_RCDATA);
  with TFileStream.Create('test.txt', fmCreate) do begin
    CopyFrom(stm, stm.Size);
  end;
end;
  
```

```

    Free;
end;
end;
end;

```

Der Compiler reserviert standardmäßig 1 MB Adressraum zusätzlich zu dem von der Anwendung beim Linken verwendeten Speicher für Ressourcen unter Kylix.

Für Linux-Entwickler sei die Unit *MultilangTranslator\_CLX.pas* zu verwenden. Es ist nicht erstaunlich, dass ein kleiner (CLX sei Dank) Umbau des Ressourcen-Zugriffs erfolgen musste, dank der Kapselung von `GetResourceString` bleibt die Schnittstelle aber stabil.

```

function EnumStringModules(Instance: Longint; Data: Pointer): Boolean;
var
    rs: TResStringRec;
    Module: HModule;
begin
    Module:= Instance;
    rs.Module:= @Module;
    with PStrData(Data)^ do begin
        rs.Identifizier:= Ident;
        Str:= System.LoadResString(@rs);
        Result:= Str = '';
    end;
end;

```

```

function TMultilangSC.GetResourceString(Ident: Integer): string;
var
    StrData: TStrData;
begin
    StrData.Ident:= Ident + LanguageOffset;
    StrData.Str:= '';
    EnumResourceModules(EnumStringModules, @StrData);
    Result:= StrData.Str;
end;

```

Mit Hilfe einer übergebenen Callback Funktion erhalten wir den Eintritt. `EnumResourceModules` führt die im Parameter `Func` angegebene Callback-Funktion für alle Instanz-Handles aus, die Ressourcen im aktuellen Programm zugeordnet sind. Die Funktion wird so lange ausgeführt, bis auch das letzte Ressourcenmodul der Anwendung aufgezählt ist oder bis die Callback-Funktion den Wert `false` zurückgibt. Sie sehen, es gibt gewisse Designprinzipien wie die Callbacks, welche so oder ähnlich in den digitalen Kammern der Kylix-Ingenieure mal begonnen haben, um einigermaßen WinAPI unabhängig zu sein und werden.

Quellen:

[http://max.kleiner.com/download/multilang\\_intro.pdf](http://max.kleiner.com/download/multilang_intro.pdf)

the component for download:

[http://www.softwareschule.ch/download/laz\\_multilang.zip](http://www.softwareschule.ch/download/laz_multilang.zip)

Max Kleiner

Beispiel auf der CD-ROM