Delphi - Kurs, 3.Teil

Eine Komponente zur Dual-Darstellung

Robert P. Michelic

Einleitung

Im dritten Teil dieser Serie soll eine Delphi-Komponente entwickelt werden, die Dualzahlen darstellen kann und bei der man Dualzahlen auch in dualer Form eingeben (einstellen) kann. Es geht uns in erster Linie darum, eine Komponente zu entwickeln, die didaktisch gut verwendbar ist, sei es in Programmen, die die Schüler entwickeln, oder in Demonstrationsprogrammen.

Um nicht bei Null anfangen zu müssen, setzen wir die Komponente aus Delphi-Komponenten zusammen. Die wesentlichen Bestandteile werden ein Panel als Träger und ein StringGrid zur Darstellung der Dualzahl sein. Dazu kommen noch Labels zur Beschriftung.

Die einzelnen Bits sollen farblich dargestellt werden - Bit ist gesetzt durch eine Farbe (rot), Bit ist nicht gesetzt durch eine andere Farbe (grün). Jedes Bit soll eine Zelle des `StringGrids` besetzen.

Die Komponente BitPanel

Wir nennen unseren Komponententyp `TBitPanel`, da er von `TPanel` abstammt. Die Felder für die zusätzlichen Komponenten sind in der folgenden Deklaration auch gleich zu finden:

```
TBitPanel=class(TPanel)
private
  FGrid:TStringGrid;
  lbBit,lbBitNr:TLabel;
end;
```

Auf diesem Panel platzieren wir ein `StringGrid` und zwei Labels, die wir im Konstruktor des Panels erzeugen:

```
constructor TBitPanel.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);
  FGrid:=TStringGrid.Create(Self);
  with FGrid do begin
     {Eigenschaften setzen - Details siehe SourceCode}
  end;
  lbBitNr:=TLabel.Create(Self);
  with lbBitNr do begin
    {Eigenschaften setzen - Details siehe SourceCode}
  end;
  lbBit:=TLabel.Create(Self);
  with lbBit do begin
    {Eigenschaften setzen - Details siehe SourceCode}
  end;
  Height:=(FGrid.DefaultRowHeight+cMarge+2)*2;
  {weitere Eigenschaften setzen - Details siehe SourceCode}
end;
```

Der Anwender soll verschiedene Eigenschaften der Komponente einstellen können, also brauchen wir entsprechende `published properties`. Wichtig ist einmal die Stellenanzahl der Dualzahl - wir nennen die Eigenschaft `BitCount`. Dann soll auch ausgewählt werden können, ob die einzelnen Bits durchnummeriert (01,02,03,...) oder mit ihrem Stellenwert (1,2,4,8,16,...) angezeigt werden sollen. Die zugehörige Eigenschaft: `UseNumbers` ist vom Typ `Boolean`. Das Wichtigste ist natürlich der tatsächlich in der Komponente „gespeicherte“ Wert - wir nennen die Eigenschaft `Value`.

Die Eigenschaft `BitCount` erfordert ein bisschen Aufmerksamkeit beim Setzen - Es muss ja die Größe der Komponenten angepasst werden:

```
procedure TBitPanel.SetBitCount(const Value: integer);
var i:integer;
    OldVal:integer;
begin
  OldVal:=Value;  {merken!}
  if Value <8 then FBitCount:=8
  else if Value >32 then FBitCount:=32
  else FBitCount := Value;
  FGrid.ColCount:=FBitCount;
  if Parent<>nil then Resize;
  if UseNumbers
  then for i:=0 to pred(FGrid.ColCount) do
     FGrid.Cells[i,rowNr]:=FormatFloat(' 00',FBitCount-i-1)
  else for i:=0 to pred(FGrid.ColCount) do
     FGrid.Cells[i,rowNr]:=FormatFloat('##0',1 shl (FBitCount-i-1));
  Value:=OldVal;
end;
```

Zuerst wird der Wert von `BitCount` zwischen 8 und 32 eingeschränkt - d.h. unsere Komponente kann Zahlen bis 32 Bit darstellen, ist somit zur Darstellung von Integer-Werten geeignet. Dann wird die Anzahl der Spalten des `StringGrids` auf den Wert `BitCount` gesetzt und `Resize` - zum Darstellen der veränderten Größenverhältnisse - aufgerufen. Zuletzt wird noch die Beschriftung der Spalten - nach Wahl des Anwenders (`Usenumbers`) - gesetzt.

`Value` wird zwischengespeichert in `OldVal` und zuletzt wieder gesetzt, das erneuert gleich die Anzeige. Da wir `Value` nicht doppelt speichern wollen, ist der gespeicherte Wert eigentlich in der Anzeige des `StringGrids` versteckt, schauen wir uns die Get- und Set-Routinen für `Value` an:

```
procedure TBitPanel.SetValue(const Value: integer);
var col,iWork:integer;
begin
  iWork:=Value;
  for col:=pred(FGrid.ColCount) downto 0 do begin
    {Anmerkung: An und für sich könnte man auch mit den Operatoren
     "mod" und "div" arbeiten. Der Operator "mod" funktioniert aber
     mit negativen Zahlen nicht, daher werden hier die Bitoperationen
     "and 1" an Stelle von "mod" und "shr"
     an Stelle von "div" verwendet.}
    if (iWork and 1)0 then FGrid.Cells[col,rowBit]:=str1
    else FGrid.Cells[col,rowBit]:=str0;
    iWork:=iWork shr 1
  end;
  if not Internal and Assigned(FOnChange) then begin
    Internal:=true;
    try
      FOnChange(Self);
    finally
      Internal:=false
    end
  end;
end;
```

Der übergebene Wert `Value` wird in seien Bits zerlegt, indem in die einzelnen Zellen des Gitters entweder die Konstante `Str1` (der String „1“) oder `str0` (der String „0“) geschrieben wird. Zuletzt wird - falls vorhanden - die Ereignisbehandlung für das Ereignis `OnChange` aufgerufen.

```
function TBitPanel.GetValue: integer;
var col:integer;
begin
  Result:=0;
  for col:=0 to pred(FGrid.ColCount) do begin
    Result:=Result shl 1;
    if HasOne(col) then Result:=Result+1
  end;
end;
```

Umgekehrt zum Setzen erhält man den aktuellen `Value` mit `GetValue`, indem in jeder Spalte die Bits geprüft werden - `HasOne` liefert `true`, wenn in der Spalte eine „1“ steht, sonst `false`.

Um die Bits auf unsere Weise (mit Farben) darstellen zu können, bearbeiten wir das Ereignis `OnDrawCell` des Stringgrids:

```
procedure TBitPanel.DrawGridCell(Sender: TObject; ACol, ARow: Integer;
  Rect: TRect; State: TGridDrawState);
begin
  if UseColor and (ARow=rowBit) then with FGrid.Canvas do begin
    if HasOne(ACol) then Brush.Color:=clRed else Brush.Color:=clGreen;
    Rectangle(Rect.Left,Rect.Top,Rect.Right,Rect.Bottom);
  end
end;
```

Wer andere Farben möchte, muss die Zuweisungen an `Brush.Color` entsprechend ändern. (Wenn `UseColor`, eine Boolean-Eigenschaft, `false` ist, wird in den Zellen 0 bzw. 1 dargestellt.)

Um auf die einzelnen Bits praktisch zugreifen zu können, deklarieren wir eine Array-Eigenschaft Bit im `public`-Teil der Deklaration von `TBitPanel`:

```
property Bit[BitNr:integer]:Boolean read GetBit write SetBit;
```

Die zugehörigen Methoden:

```
function TBitPanel.GetBit(BitNr: integer): Boolean;
begin
  if BitNr<FGrid.ColCount then Result:=HasOne(FGrid.ColCount-BitNr-1)
  else Result:=false;
end;

procedure TBitPanel.SetBit(BitNr: integer; const Value: Boolean);
begin
  if BitNr<FGrid.ColCount then begin
    if Value then FGrid.Cells[FGrid.ColCount-BitNr-1,rowBit]:=str1
    else FGrid.Cells[FGrid.ColCount-BitNr-1,rowBit]:=str0;
    if not Internal and Assigned(FOnChange) then begin
      Internal:=true;
      try
        FOnChange(Self);
      finally
        Internal:=false
      end
    end;
  end
end;
```

Auch in `SetBit` wird wieder - falls deklariert - die Ereignisbehandlung für `OnChange` aufgerufen.

Ein bisschen Spielerei ist die Bearbeitung von Größenänderungen bzw. das Ausrichten der einzelnen Komponenten. Zuerst passen wir die Größe des `Stringgrids` den Erfordernissen (Schriftart!) an, dann die Breite und Höhe des Panels selber. Zuletzt richten wir noch die beiden Labels passend aus. Die Konstante `cMarge` gibt uns einen einheitlichen Rand um die einzelnen Komponenten.

Um die Komponente nicht zu überladen, wurden in der vorliegenden Version diese Anpassungen nur angedeutet, z.B. wird auf eine Änderung der Schriftart nur nach einer Größenänderung und auch nur in der Breite reagiert. Bei Bedarf aber lassen sich diese zusätzlichen Eigenschaften leicht ergänzen.

```
procedure TBitPanel.Resize;
begin
  FGrid.DefaultColWidth:=Canvas.TextWidth(' 15  ');
  FGrid.ClientWidth:=(FGrid.DefaultColWidth+1)*FGrid.ColCount;
  FGrid.ClientHeight:=(FGrid.DefaultRowHeight+1)*FGrid.RowCount;
  FGrid.Left:=cMarge*2+lbBitNr.Width;
  Width:=FGrid.Left+FGrid.Width+cMarge;
  Height:=cMarge*2+FGrid.Height;
  lbBitNr.Top:=cMarge+(FGrid.DefaultRowHeight-lbBitNr.Height) div 2 ;
  lbBit.Top:=cMarge+(FGrid.DefaultRowHeight*3-lbBit.Height) div 2;
  inherited Resize;
end;
```

Um die Bits auch per Mausklick umschalten zu können, behandeln wir das Ereignis `OnDblClick` mit der Ereignisbehandlungsroutine `ToggleBit`:

```
procedure TBitPanel.ToggleBit(Sender:TObject);
begin
  if HasOne(FGrid.Col) then FGrid.Cells[FGrid.Col,rowBit]:=str0
  else FGrid.Cells[FGrid.Col,rowBit]:=str1;
  if Assigned(FOnChange) then begin
    Internal:=true;
    try
      FOnChange(Self);
    finally
      Internal:=false
    end
  end;
end;
```

Die restlichen Details der Deklaration von `TBitPanel` findet man im Anhang.

Wenn wir `TBitPanel` in die Komponentenpalette integrieren möchten, müssen wir die Komponente registrieren und ein entsprechendes Package kompilieren, das fertige Package ist neben dem Programmlisting beim Autor erhältlich (*rpmsoft@via.at*).

Als Anwendungsbeispiel wird die Komponente eine einem Programm zur Demonstration der Addition von Dualzahlen verwendet:

Mit jeweils einer Komponente `TBitPanel` werden die beiden Summanden dargestellt, eine weitere zeigt den notwendigen Übertrag für jedes Bit, die vierte stellt die Summe dar. Man kann die Addition in Einzelschritten (bitweise) ablaufen lassen und dabei genau verfolgen, wie die Bits addiert werden.

Das zugehörige Programm befindet sich im Anhang.

Anhang: Programmlistings

Die Unit uBitPanel enthält die Deklaration der Komponente TBitPanel:

```
unit UBitPanel;

interface

uses Windows,Classes,StdCtrls,ExtCtrls,Grids;

type
  TBitPanel=class(TPanel)
  private
    FBitCount:integer;
    FUseColor,FUseNumbers:Boolean;
    FGrid:TStringGrid;
    FOnChange:TNotifyEvent;
    lbBit,lbBitNr:TLabel;
    Internal:Boolean;
    procedure SetBitCount(const BitCnt: integer);
    function GetValue: integer;
    procedure SetValue(const Value: integer);
    function HasOne(col:integer):Boolean;
    procedure ToggleBit(Sender:TObject);
    procedure DrawGridCell(Sender: TObject; ACol, ARow: Longint; Rect:
TRect; State: TGridDrawState);
    procedure SetOnChange(const Value: TNotifyEvent);
    procedure SetUseColor(const Value: Boolean);
    function GetBit(BitNr: integer): Boolean;
    procedure SetBit(BitNr: integer; const Value: Boolean);
    procedure SetUseNumbers(const Value: Boolean);
```

```
    protected
      procedure Resize; override;
    public
      constructor Create(AOwner:TComponent); override;
      procedure CreateWnd; override;
      destructor Destroy; override;
      property Bit[BitNr:integer]:Boolean read GetBit write SetBit;
    published
      property BitCount:integer read FBitCount write SetBitCount;
      property Value:integer read GetValue write SetValue;
      property UseColor:Boolean read FUseColor write SetUseColor;
      property UseNumbers:Boolean read FUseNumbers write SetUseNumbers;
      property OnChange:TNotifyEvent read FOnChange write SetOnChange;
    end;

procedure Register;

implementation

uses SysUtils,Graphics;

const
  str0=' 0 ';
  str1=' 1 ';
  rowNr=0;
  rowBit=1;
  cMarge=4;

procedure Register;
begin
  RegisterComponents('Kurs',[TBitPanel]);
end;

{ TBitPanel }

constructor TBitPanel.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);
  FGrid:=TStringGrid.Create(Self);
  with FGrid do begin
    Parent:=Self;
    FixedCols:=0;
    FixedRows:=1;
    ColCount:=8;
    RowCount:=2;
    Top:=cMarge;
    Left:=100;
    Options:=[goFixedVertLine, goFixedHorzLine, goVertLine,
goHorzLine];
    ScrollBars:=ssNone;
    OnDblClick:=ToggleBit;
    OnDrawCell:=DrawGridCell;
  end;
  lbBitNr:=TLabel.Create(Self);
  with lbBitNr do begin
    Parent:=Self;
    Left:=cMarge;
    Top:=cMarge;
    Caption:='Bit Nr.:';
  end;
  lbBit:=TLabel.Create(Self);
  with lbBit do begin
    Parent:=Self;
    Left:=cMarge;
    Top:=25;
    Caption:='Bit:';
  end;
  Height:=(FGrid.DefaultRowHeight+cMarge+2)*2;
  Value:=0;
  BitCount:=8;
end;

procedure TBitPanel.CreateWnd;
begin
  inherited CreateWnd;
  Resize;
end;

destructor TBitPanel.Destroy;
begin
  FGrid.Free;
  FGrid:=nil;
  lbBitNr.Free;
  lbBitNr:=nil;
  lbBit.Free;
  lbBit:=nil;
  inherited Destroy;
end;

procedure TBitPanel.Resize;
begin
  FGrid.DefaultColWidth:=Canvas.TextWidth(' 15  ');
  FGrid.ClientWidth:=(FGrid.DefaultColWidth+1)*FGrid.ColCount;
  FGrid.ClientHeight:=(FGrid.DefaultRowHeight+1)*FGrid.RowCount;
  FGrid.Left:=cMarge*2+lbBitNr.Width;
  Width:=FGrid.Left+FGrid.Width+cMarge;
  Height:=cMarge*2+FGrid.Height;
  lbBitNr.Top:=cMarge+(FGrid.DefaultRowHeight-lbBitNr.Height) div 2 ;
  lbBit.Top:=cMarge+(FGrid.DefaultRowHeight*3-lbBit.Height) div 2;
  inherited Resize;
end;

procedure TBitPanel.SetBitCount(const BitCnt: integer);
var i:integer;
    OldVal:integer;
begin
  OldVal:=Value;  {merken!}
  if BitCnt<8 then FBitCount:=8
  else if BitCnt>32 then FBitCount:=32
  else FBitCount := BitCnt;
  FGrid.ColCount:=FBitCount;
  if Parentnil then Resize;
  if UseNumbers then for i:=0 to pred(FGrid.ColCount) do
FGrid.Cells[i,rowNr]:=FormatFloat(' 00',FBitCount-i-1)
  else for i:=0 to pred(FGrid.ColCount) do
FGrid.Cells[i,rowNr]:=FormatFloat('##0',1 shl (FBitCount-i-1));
  Value:=OldVal;
end;

function TBitPanel.GetValue: integer;
var col:integer;
begin
  Result:=0;
  for col:=0 to pred(FGrid.ColCount) do begin
    Result:=Result shl 1;
    if HasOne(col) then Result:=Result+1
  end;
end;

procedure TBitPanel.SetValue(const Value: integer);
var col,iWork:integer;
begin
  iWork:=Value;
  for col:=pred(FGrid.ColCount) downto 0 do begin
    {Anmerkung: An und für sich könnte man auch mit den Operatoren
     "mod" und "div" arbeiten. Der Operator "mod" funktioniert aber
     mit negativen Zahlen nicht, daher werden hier die Bitoperationen
     "and 1" an Stelle von "mod" und "shr"
     an Stelle von "div" verwendet.}
    if (iWork and 1)0 then FGrid.Cells[col,rowBit]:=str1
    else FGrid.Cells[col,rowBit]:=str0;
    iWork:=iWork shr 1
  end;
  if not Internal and Assigned(FOnChange) then begin
    Internal:=true;
    try
      FOnChange(Self);
    finally
      Internal:=false
    end
  end;
end;

function TBitPanel.HasOne(col:integer):Boolean;
begin
  Result:=Pos('1',FGrid.Cells[col,rowBit])>0
end;

procedure TBitPanel.ToggleBit(Sender:TObject);
begin
  if HasOne(FGrid.Col) then FGrid.Cells[FGrid.Col,rowBit]:=str0
  else FGrid.Cells[FGrid.Col,rowBit]:=str1;
  if Assigned(FOnChange) then begin
    Internal:=true;
    try
      FOnChange(Self);
    finally
      Internal:=false
    end
  end;
end;

procedure TBitPanel.SetOnChange(const Value: TNotifyEvent);
begin
  FOnChange := Value;
end;

procedure TBitPanel.SetUseColor(const Value: Boolean);
begin
```

```
    FUseColor := Value;
    Repaint;
  end;

  procedure TBitPanel.DrawGridCell(Sender: TObject; ACol, ARow: Integer;
    Rect: TRect; State: TGridDrawState);
  begin
    if UseColor and (ARow=rowBit) then with FGrid.Canvas do begin
      if HasOne(ACol) then Brush.Color:=clRed else Brush.Color:=clGreen;
      Rectangle(Rect.Left,Rect.Top,Rect.Right,Rect.Bottom);
    end
  end;

  function TBitPanel.GetBit(BitNr: integer): Boolean;
  begin
    if BitNr<FGrid.ColCount then Result:=HasOne(FGrid.ColCount-BitNr-1)
    else Result:=false;
  end;

  procedure TBitPanel.SetBit(BitNr: integer; const Value: Boolean);
  begin
    if BitNr<FGrid.ColCount then begin
      if Value then FGrid.Cells[FGrid.ColCount-BitNr-1,rowBit]:=str1
      else FGrid.Cells[FGrid.ColCount-BitNr-1,rowBit]:=str0;
      if not Internal and Assigned(FOnChange) then begin
        Internal:=true;
        try
          FOnChange(Self);
        finally
          Internal:=false
        end
      end;
    end
  end;

  procedure TBitPanel.SetUseNumbers(const Value: Boolean);
  var i:integer;
  begin
    FUseNumbers := Value;
    if Value then for i:=0 to pred(FGrid.ColCount) do
  FGrid.Cells[i,rowNr]:=FormatFloat(' 00',FBitCount-i-1)
    else for i:=0 to pred(FGrid.ColCount) do
  FGrid.Cells[i,rowNr]:=FormatFloat('##0',1 shl (FBitCount-i-1));
  end;

  end.
```

Die `Unit uAdd` gehört zum Addierprogramm, ein Anwendungsbeispiel der Komponente `TBitPanel`. Wichtig: Diese `Unit` hängt von der `Unit uBitPanel`, die die Deklaration von `TBitPanel` enthält, ab. Das zugehörige Formular kann in Delphi nur angezeigt werden, wenn das Package mit der Komponente `TBitPanel` auch installiert ist!

```
  unit UAdd;

  interface

  uses
    Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
  Dialogs,
    StdCtrls, ExtCtrls, UBitPanel, ComCtrls;

  type
    TfrmAdd = class(TForm)
      bp1: TBitPanel;
      bp2: TBitPanel;
      bpC: TBitPanel;
      bpS: TBitPanel;
      edS1: TEdit;
      edS2: TEdit;
      edSum: TEdit;
      Panel1: TPanel;
      Label1: TLabel;
      btnSingleStep: TButton;
      btnAdd: TButton;
      btnClose: TButton;
      Label2: TLabel;
      Label3: TLabel;
      progBar: TProgressBar;
      procedure btnCloseClick(Sender: TObject);
      procedure edS12Change(Sender: TObject);
      procedure btnSingleStepClick(Sender: TObject);
      procedure FormCreate(Sender: TObject);
      procedure btnAddClick(Sender: TObject);
      procedure bpSChange(Sender: TObject);
      procedure bp1Change(Sender: TObject);
      procedure bp2Change(Sender: TObject);
    private
      { Private-Deklarationen }
      FStepCount:integer;
      procedure SetStepCount(const Value:Integer);
      procedure AddBit(BitNr:integer);
      property StepCount:integer read FStepCount write SetStepCount;
    public
      { Public-Deklarationen }
    end;

  var
    frmAdd: TfrmAdd;

  implementation

  {$R *.DFM}

  procedure TFrmAdd.SetStepCount(const Value:Integer);
  begin
    FStepCount:=Value;
    progBar.Position:=9-Value;
  end;

  procedure TFrmAdd.AddBit(BitNr:integer);
  begin
    bpS.Bit[BitNr]:=bp1.Bit[BitNr] xor bp2.Bit[BitNr] xor
  bpC.Bit[BitNr];
    bpC.Bit[BitNr+1]:=(bp1.Bit[BitNr] and bp2.Bit[BitNr])
                      or ((bp1.Bit[BitNr] or bp2.Bit[BitNr]) and
  bpC.Bit[BitNr]);
  end;

  procedure TFrmAdd.btnCloseClick(Sender: TObject);
  begin
    Close
  end;

  procedure TFrmAdd.edS12Change(Sender: TObject);
  begin
    try
      bp1.Value:=StrToInt(edS1.Text);
      bp2.Value:=StrToInt(edS2.Text);
    except
    end;
    bpC.value:=0;
    bpS.Value:=0;
    StepCount:=0;
  end;

  procedure TFrmAdd.btnSingleStepClick(Sender: TObject);
  begin
    if StepCount<bp1.BitCount then begin
      AddBit(StepCount);
      StepCount:=StepCount+1;
    end;
    if StepCount=bp1.BitCount then begin
      bpS.Bit[StepCount]:=bpC.Bit[StepCount];
      StepCount:=StepCount+1;
    end;
  end;

  procedure TFrmAdd.FormCreate(Sender: TObject);
  begin
    edS1.Text:='0';
    edS2.Text:='0';
    edSum.Text:='0'
  end;

  procedure TFrmAdd.btnAddClick(Sender: TObject);
  begin
    while StepCount<bp1.BitCount do btnSingleStepClick(nil)
  end;

  procedure TFrmAdd.bpSChange(Sender: TObject);
  begin
    edSum.Text:=IntToStr(bpS.Value)
  end;

  procedure TFrmAdd.bp1Change(Sender: TObject);
  begin
    edS1.Text:=IntToStr(bp1.Value)
  end;

  procedure TFrmAdd.bp2Change(Sender: TObject);
  begin
    edS2.Text:=IntToStr(bp2.Value)
  end;

  end.
```