FastReport 4 Developer's manual

Manual v1.2.0

Table of Contents

FastReport Class Hierarchy 1

Writing Custom Report Components 7

Writing Custom Common Controls 10

Event Handler Description 13

Component Registration in Script System 14

Writing Component Editors 15

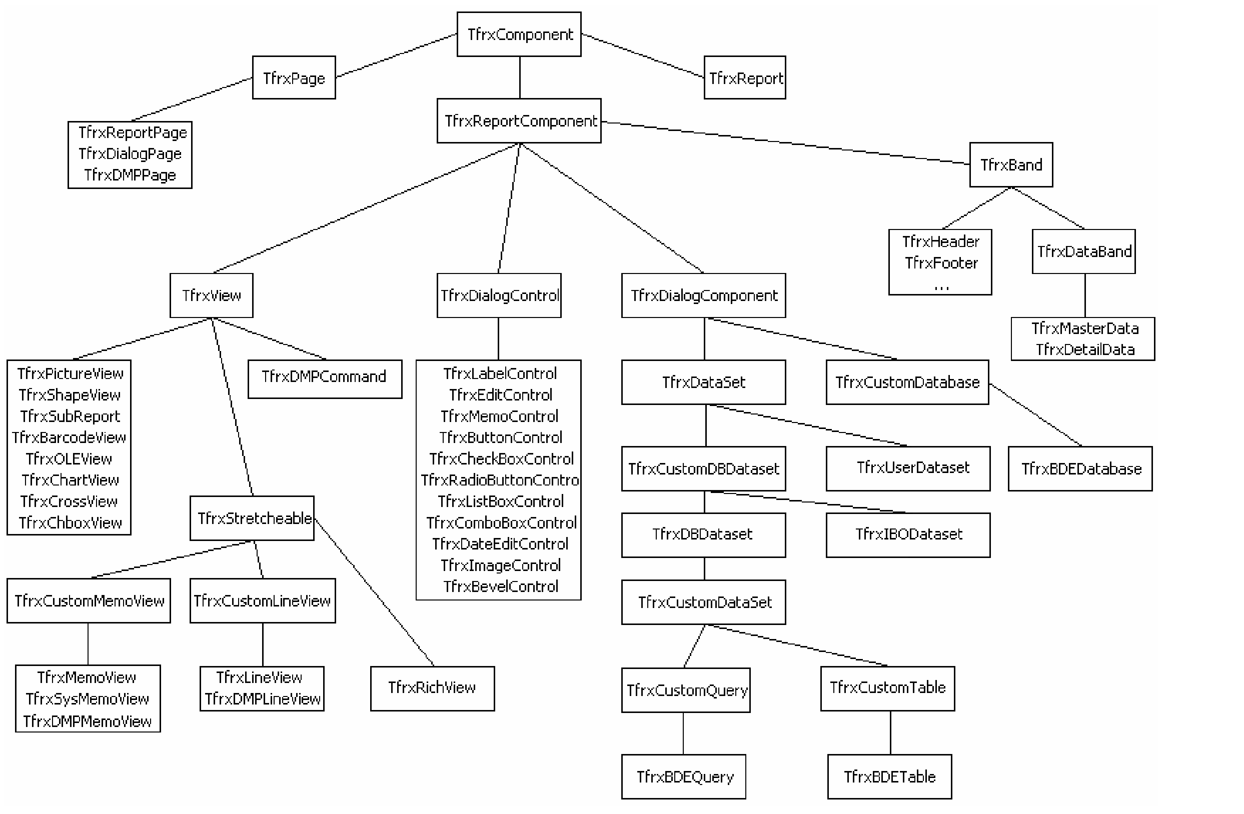
Writing Property Editors 17

Writing Custom DB Engines 23

Using Custom Functions in a Report 37

Writing Custom Wizards 40

FastReport Class Hierarchy



"**TfrxComponent**" is the base class for all FastReport components. Objects of this type have attributes such as “coordinates”, “size”, “font” and “visibility” and have lists of subordinate objects. The class also has methods which allow for the saving and restoration of the object state to or from a stream.

```
TfrxComponent = class(TComponent)
protected
  procedure SetParent(AParent: TfrxComponent); virtual;
  procedure SetLeft(Value: Extended); virtual;
  procedure SetTop(Value: Extended); virtual;
  procedure SetWidth(Value: Extended); virtual;
  procedure SetHeight(Value: Extended); virtual;
  procedure SetFont(Value: TFont); virtual;
  procedure SetParentFont(Value: Boolean); virtual;
  procedure SetVisible(Value: Boolean); virtual;
  procedure FontChanged(Sender: TObject); virtual;
public
  constructor Create(AOwner: TComponent); override;
  procedure Assign(Source: TPersistent); override;
  procedure Clear; virtual;
  procedure CreateUniqueName;
  procedure LoadFromStream(Stream: TStream); virtual;
  procedure SaveToStream(Stream: TStream); virtual;
  procedure SetBounds(ALeft, ATop, AWidth, AHeight: Extended);
  function FindObject(const AName: String): TfrxComponent;
  class function GetDescription: String; virtual;
```

```
    property Objects: TList readonly;
    property AllObjects: TList readonly;
    property Parent: TfrxComponent;
    property Page: TfrxPage readonly;
    property Report: TfrxReport readonly;
    property IsDesigning: Boolean;
    property IsLoading: Boolean;
    property IsPrinting: Boolean;
    property BaseName: String;

    property Left: Extended;
    property Top: Extended;
    property Width: Extended;
    property Height: Extended;
    property AbsLeft: Extended readonly;
    property AbsTop: Extended readonly;

    property Font: TFont;
    property ParentFont: Boolean;
    property Restrictions: TfrxRestrictions;
    property Visible: Boolean;
  end;
```

- *Clear*	clears object contents and deletes all its child objects
- *CreateUniqueName*	creates unique name for object placed into report
- *LoadFromStream*	loads object and all its child objects from stream
- *SaveToStream*	saves object to stream
- *SetBounds*	sets object coordinates and size
- *FindObject*	searches for object with specified name among child objects
- *GetDescription*	returns object’s description

The following methods are called when modifying the corresponding properties. If additional handling is needed, you can override them:

- SetParent
- SetLeft
- SetTop
- SetWidth
- SetHeight
- SetFont
- SetParentFont
- SetVisible
- FontChanged

The following properties are defined in the “TfrxComponent” class:

- *Objects*	list of child objects
- *AllObjects*	list of all subordinate objects
- *Parent*	link to parent object
- *Page*	link to report page on which object resides
- *Report*	link to report in which object appears
- *IsDesigning*	“True” if designer is running
- *IsLoading*	“True” if object is being loaded from stream
- *IsPrinting*	“True” if object is being printed out
- *BaseName*	object base name; this value is used in *“CreateUniqueName”* method
- *Left*	object X-coordinate (relative to parent)

- *Top*	object Y-coordinate (relative to parent)
- *Width*	object width
- *Height*	object height
- *AbsLeft*	object absolute X-coordinate
- *AbsTop*	object absolute Y-coordinate
- *Font*	object font
- *ParentFont*	if "True" then uses parent object font settings
- *Restrictions*	set of flags which restricts one or more object operations
- *Visible*	object visibility

The next base class is **"TfrxReportComponent"**. Objects of this type can be placed into a report design. This class has the "Draw" method for drawing the object and also the "BeforePrint/GetData/AfterPrint" methods, which are called when the report is running.

```
TfrxReportComponent = class(TfrxComponent)
public
  procedure Draw(Canvas: TCanvas;
                 ScaleX, ScaleY, OffsetX, OffsetY: Extended); virtual;
abstract;
  procedure BeforePrint; virtual;
  procedure GetData; virtual;
  procedure AfterPrint; virtual;
  function GetComponentText: String; virtual;
  property OnAfterPrint: TfrxNotifyEvent;
  property OnBeforePrint: TfrxNotifyEvent;
end;
```

The "Draw" method is called to draw the object, with parameters:

- *Canvas*	*on* canvas
- *Scale*	scaling along the X- and Y-axes
- *Offset*	offset relative to canvas edges

The "BeforePrint" method is called immediately before object handling (during the report building process). This method saves the object state.

The "GetData" method is called to load data into the object.

The "AfterPrint" method is called after object handling. This method restores the object state.

"**TfrxDialogComponent**" is the base class for writing non-visual components that can be used in dialogue forms in a report.

```
TfrxDialogComponent = class(TfrxReportComponent)
public
  property Bitmap: TBitmap;
  property Component: TComponent;
published
  property Left;
  property Top;
end;
```

"TfrxDialogControl" is the base class for writing common controls that can be used in dialogue forms in a report. This class has a large number of general properties and events shared by most of the common controls.

```
TfrxDialogControl = class(TfrxReportComponent)
protected
  procedure InitControl(AControl: TControl);
public
  property Caption: String;
  property Color: TColor;
  property Control: TControl;
  property OnClick: TfrxNotifyEvent;
  property OnDblClick: TfrxNotifyEvent;
  property OnEnter: TfrxNotifyEvent;
  property OnExit: TfrxNotifyEvent;
  property OnKeyDown: TfrxKeyEvent;
  property OnKeyPress: TfrxKeyPressEvent;
  property OnKeyUp: TfrxKeyEvent;
  property OnMouseDown: TfrxMouseEvent;
  property OnMouseMove: TfrxMouseMoveEvent;
  property OnMouseUp: TfrxMouseEvent;
published
  property Left;
  property Top;
  property Width;
  property Height;
  property Font;
  property ParentFont;
  property Enabled: Boolean;
  property Visible;
end;
```

When writing your own custom control, you should inherit from this class, move the required properties to the “published” section, and then add new properties for your common control. The writing of custom controls will be discussed in detail in the next chapter.

"**TfrxView**" is the base class for most components that can be placed on the report design page. Objects of this class have attributes such as “Frame” and “Filling” and can also be connected to a data source. Most FastReport standard objects inherit from this class.

```
TfrxView = class(TfrxReportComponent)
protected
  FX, FY, FX1, FY1, FDX, FDY, FFrameWidth: Integer;
  FScaleX, FScaleY: Extended;
  FCanvas: TCanvas;
  procedure BeginDraw(Canvas: TCanvas; ScaleX, ScaleY, OffsetX, OffsetY:
Extended); virtual;
  procedure DrawBackground;
  procedure DrawFrame;
  procedure DrawLine(x, y, x1, y1, w: Integer);
public
  function IsDataField: Boolean;
  property BrushStyle: TBrushStyle;
  property Color: TColor;
  property DataField: String;
  property DataSet: TfrxDataSet;
  property Frame: TfrxFrame;
published
  property Align: TfrxAlign;
  property Printable: Boolean;
  property ShiftMode: TfrxShiftMode;
  property TagStr: String;
  property Left;
```

```
    property Top;
    property Width;
    property Height;
    property Restrictions;
    property Visible;
    property OnAfterPrint;
    property OnBeforePrint;
  end;
```

The following methods are defined in this class:

- *BeginDraw*	called by the "Draw" method; calculates integer-valued coordinates and size of drawing area. Calculated values are exposed as FX, FY, FX1, FY1, FDX and FDY variables. Frame width (exposed as FFrameWidth) is also calculated
- *DrawBackground*	draws object background
- *DrawFrame*	draws object frame
- *DrawLine*	draws line with specified coordinates and width
- IsDataField	returns "True" if DataSet and DataField properties contain non-empty values

One can refer to the following properties after calling the "BeginDraw" method:

- FX, FY, FX1, FY1, FDX, FDY, FFrameWidth	the object frame coordinates, sizes and width calculated using the Scale and Offset parameters
- FscaleX, FScaleY	X & Y-scaling, which are copies of the ScaleX and ScaleY parameters from the Draw method
- FCanvas	the canvas, which is a copy of the Canvas parameter from the Draw method

The class defines the following properties, which are found in most report objects:

- *BrushStyle*	object fill style
- *Color*	object fill color
- *DataField*	name of data field to which object is connected
- *DataSet*	data source
- *Frame*	object frame
- *Align*	object alignment relative to its parent
- *Printable*	defines whether given object can be printed out
- *ShiftMode*	shift mode of object in cases when a stretchable object is placed above it
- *TagStr*	field for storing user information

"**TfrxStretcheable**" is the base class for writing components which modify their height depending on the data placed in them.

```
  TfrxStretcheable = class(TfrxView)
  public
    function CalcHeight: Extended; virtual;
    function DrawPart: Extended; virtual;
    procedure InitPart; virtual;
```

```
published
  property StretchMode: TfrxStretchMode;
end;
```

Objects of this class can be stretched vertically and also split into smaller pieces when the objects don't have sufficient width on the output page. The objects are split enough times to fit all of their data into the available space.

The following methods are defined in this class:

- *CalcHeight*	calculates and returns object height according to the data placed in it
- *InitPart*	called before object splitting
- *DrawPart*	draws next data chunk placed in object "Return value" is the amount of unused space where it was impossible to display data

Writing Custom Report Components

FastReport has a large number of components that can be placed on a report design page. They are: text, picture, line, geometrical figure, OLE, rich text, bar code, diagram etc. You can also write your own custom component and then attach it to FastReport.

FastReport has several classes from which components can be inherited. For more details, see “FastReport Class Hierarchy”. The TfrxView class is of primary interest, since most report components are inherited from it.

As a minimum the “Draw” method in the TfrxReportComponent base class should be defined.

```
procedure Draw(Canvas: TCanvas;
               ScaleX, ScaleY, OffsetX, OffsetY: Extended); virtual;
```

This method is called when the component is painted in the designer, in the preview window and during output printing. TfrxView overrides this method for drawing the object frame and background. This method should draw the component's contents on the “Canvas” drawing surface. The object coordinates and sizes are stored in the “AbsLeft”, “AbsTop” and “Width”, “Height” properties respectively.

The “ScaleX” and “ScaleY” parameters define the object scaling in the X-axis and Y-axis respectively. These parameters equal 1 at 100% zoom and can change if the user modifies zooming either in the designer or in the preview window. The “OffsetX” and “OffsetY” parameters shift the object along the X-axis and Y-axis. So, taking all these parameters into account the upper left corner coordinate will be:

```
X := Round(AbsLeft * ScaleX + OffsetX);
Y := Round(AbsTop * ScaleY + OffsetY);
```

To simplify operations with coordinates, the “BeginDraw” method (with parameters similar to “Draw”) is defined in the “TfrxView” class.

```
procedure BeginDraw(Canvas: TCanvas;
                    ScaleX, ScaleY, OffsetX, OffsetY: Extended); virtual;
```

This method should be called in the first line of the “Draw” method. It transforms the coordinates into FX, FY, FX1, FY1, FDX, FDY and FFrameWidth integer values, which can be used later in TCanvas methods. It also copies Canvas, ScaleX and ScaleY values into the FCanvas, FScaleX and FScaleY variables, which can be referred to in any class method.

There are also two methods in the TfrxView class for drawing object backgrounds and frames.

```
procedure DrawBackground;
procedure DrawFrame;
```

The BeginDraw method should be called before calling these two methods.

Let's look at how to create a component which will display an arrow.

```
type
  TfrxArrowView = class(TfrxView)
  public
    { we should override only two methods }
    procedure Draw(Canvas: TCanvas;
                   ScaleX, ScaleY, OffsetX, OffsetY: Extended); override;
    class function GetDescription: String; override;
  published
    { place required properties in the published section }
    property BrushStyle;
    property Color;
    property Frame;
  end;

class function TfrxArrowView.GetDescription: String;
begin
  { component description will be displayed next to its icon in toolbar }
  Result := 'Arrow object';
end;

procedure TfrxArrowView.Draw(Canvas: TCanvas;
                             ScaleX, ScaleY, OffsetX, OffsetY: Extended);
begin
  { call this method to transform coordinates }
  BeginDraw(Canvas, ScaleX, ScaleY, OffsetX, OffsetY);
  with Canvas do
  begin
    { set colors }
    Brush.Color := Color;
    Brush.Style := BrushStyle;
    Pen.Width := FFrameWidth;
    Pen.Color := Frame.Color;
    { draw arrow }
    Polygon(
      [Point(FX, FY + FDY div 4),
      Point(FX + FDX * 38 div 60, FY + FDY div 4),
      Point(FX + FDX * 38 div 60, FY),
      Point(FX1, FY + FDY div 2),
      Point(FX + FDX * 38 div 60, FY1),
      Point(FX + FDX * 38 div 60, FY + FDY * 3 div 4),
      Point(FX, FY + FDY * 3 div 4)]);
  end;
end;

{ registration }
var
  Bmp: TBitmap;

initialization
  Bmp := TBitmap.Create;
  Bmp.LoadFromResourceName(hInstance, 'frxArrowView');
  frxObjects.RegisterObject(TfrxArrowView, Bmp);

finalization
  { delete from list of available components }
  frxObjects.Unregister(TfrxArrowView);
  Bmp.Free;

end.
```

To create a component which displays data from a DB move the DataSet and DataField properties into the “published” section and then override the “GetData” method. Let's look at this by using the TfrxCheckBoxView standard component as an example.

The **“TfrxCheckBoxView”** component can be connected to a DB field using the “DataSet” and “DataField” properties, which are declared in the TfrxView base class. This component also has the “Expression” property which can hold an expression. As soon as the expression has been calculated the result is placed in the “Checked” property. The component displays a cross when “Checked” is “True.” Below are the most important parts of the component’s definition.

```
TfrxCheckBoxView = class(TfrxView)
private
  FChecked: Boolean;
  FExpression: String;
  procedure DrawCheck(ARect: TRect);
public
  procedure Draw(Canvas: TCanvas;
               ScaleX, ScaleY, OffsetX, OffsetY: Extended); override;
  procedure GetData; override;
published
  property Checked: Boolean read FChecked write FChecked default True;
  property DataField;
  property DataSet;
  property Expression: String read FExpression write FExpression;
end;

procedure TfrxCheckBoxView.Draw(Canvas: TCanvas;
                                ScaleX, ScaleY, OffsetX, OffsetY:
Extended);
begin
  BeginDraw(Canvas, ScaleX, ScaleY, OffsetX, OffsetY);

  DrawBackground;
  DrawCheck(Rect(FX, FY, FX1, FY1));
  DrawFrame;
end;

procedure TfrxCheckBoxView.GetData;
begin
  inherited;
  if IsDataField then
    FChecked := DataSet.Value[DataField]
  else if FExpression <> '' then
    FChecked := Report.Calc(FExpression);
end;
```

Writing Custom Common Controls

FastReport contains a set of common controls which can be placed on dialogue forms inside reports. They are as follows:

```
TfrxLabelControl
TfrxEditControl
TfrxMemoControl
TfrxButtonControl
TfrxCheckBoxControl
TfrxRadioButtonControl
TfrxListBoxControl
TfrxComboBoxControl
TfrxDateEditControl
TfrxImageControl
TfrxBevelControl
TfrxPanelControl
TfrxGroupBoxControl
TfrxBitBtnControl
TfrxSpeedButtonControl
TfrxMaskEditControl
TfrxCheckListBoxControl
```

These controls correspond to the Delphi component palette standard controls. If the standard functionality is not sufficient then you can create your own common controls for use in your reports.

The base class for all common controls is the **“TfrxDialogControl”** class, declared in the frxClass file:

```
TfrxDialogControl = class(TfrxReportComponent)
protected
  procedure InitControl(AControl: TControl);
public
  constructor Create(AOwner: TComponent); override;
  destructor Destroy; override;
  class function GetDescription: String; virtual;
  property Caption: String;
  property Color: TColor;
  property Control: TControl;
  property OnClick: TfrxNotifyEvent;
  property OnDblClick: TfrxNotifyEvent;
  property OnEnter: TfrxNotifyEvent;
  property OnExit: TfrxNotifyEvent;
  property OnKeyDown: TfrxKeyEvent;
  property OnKeyPress: TfrxKeyPressEvent;
  property OnKeyUp: TfrxKeyEvent;
  property OnMouseDown: TfrxMouseEvent;
  property OnMouseMove: TfrxMouseMoveEvent;
  property OnMouseUp: TfrxMouseEvent;
published
  property Left;
  property Top;
  property Width;
  property Height;
  property Font;
  property ParentFont;
  property Enabled: Boolean;
  property Visible;
```

```
end;
```

To create your own control you should inherit from this class and override at least the constructor and the “GetDescription” method. It will be necessary to create the common control and initialize it using the “InitControl” method in the constructor. The GetDescription method is for returning a description of the common control. As you can see the TfrxDialogControl class already has a large number of properties and methods in the public section. Move properties and events into the “published” section of your common control as required and also create new properties that are specific to your control.

Common control registration and deletion is performed by using the frxObjects global object methods declared in the frxDsgnIntf file:

```
frxObjects.RegisterObject(ClassRef: TfrxComponentClass;
                          ButtonBmp: TBitmap);
frxObjects.Unregister(ClassRef: TfrxComponentClass);
```

During registration you should specify the control class name and its picture. The ButtonBmp size should be 16x16 pixels.

Let's look at an example of a common control that simplifies the functionality of the standard Delphi TBitBtn control.

```
uses frxClass, frxDsgnIntf, Buttons;

type
  TfrxBitBtnControl = class(TfrxDialogControl)
  private
    FButton: TBitBtn;
    procedure SetKind(const Value: TBitBtnKind);
    function GetKind: TBitBtnKind;
  public
    constructor Create(AOwner: TComponent); override;
    class function GetDescription: String; override;
    property Button: TBitBtn read FButton;
  published
    { add new properties }
    property Kind: TBitBtnKind read GetKind
                               write SetKind default bkCustom;
    { following properties are already declared in parent class }
    property Caption;
    property OnClick;
    property OnEnter;
    property OnExit;
    property OnKeyDown;
    property OnKeyPress;
    property OnKeyUp;
    property OnMouseDown;
    property OnMouseMove;
    property OnMouseUp;
  end;

constructor TfrxBitBtnControl.Create(AOwner: TComponent);
begin
  { default constructor }
  inherited;
  { create required common control }
  FButton := TBitBtn.Create(nil);
  FButton.Caption := 'BitBtn';
  { initialize it }
  InitControl(FButton);

  { set default size }
  Width := 75;
  Height := 25;
end;

class function TfrxBitBtnControl.GetDescription: String;
begin
  Result := 'BitBtn control';
end;

procedure TfrxBitBtnControl.SetKind(const Value: TBitBtnKind);
begin
  FButton.Kind := Value;
end;

function TfrxBitBtnControl.GetKind: TBitBtnKind;
begin
  Result := FButton.Kind;
end;

var
  Bmp: TBitmap;

initialization
  Bmp := TBitmap.Create;
  { load picture from resource;
    it should have already been placed there, of course }
  Bmp.LoadFromResourceName(hInstance, 'frxBitBtnControl');
  frxObjects.RegisterObject(TfrxBitBtnControl, Bmp);

finalization
  frxObjects.Unregister(TfrxBitBtnControl);
  Bmp.Free;

end.
```

Event Handler Description

How should a new event handler be defined if it does not already belong to the base class? Let's look at this using the “TfrxEditControl” common control as an example:

```
TfrxEditControl = class(TfrxDialogControl)
private
  FEdit: TEdit;
  { new event }
  FOnChange: TfrxNotifyEvent;
  procedure DoOnChange(Sender: TObject);
  ...
public
  constructor Create(AOwner: TComponent); override;
  ...
published
  { new event }
  property OnChange: TfrxNotifyEvent read FOnChange write FOnChange;
  ...
end;

constructor TfrxEditControl.Create(AOwner: TComponent);
begin
  ...
  { connect our handler }
  FEdit.OnChange := DoOnChange;
  InitControl(FEdit);
  ...
end;

procedure TfrxEditControl.DoOnChange(Sender: TObject);
begin
  { call event handler }
  if Report <> nil then
    Report.DoNotifyEvent(Sender, FOnChange);
end;
```

It is important to note that the event handler in FastReport is a procedure defined in the report script. The TfrxNotifyEvent type is declared as String[63] and in FastReport the link to the handler is a string containing its name. This is unlike the Delphi TNotifyEvent type, in which it is a method address.

Component Registration in Script System

To refer to our component from a script, the class, its properties and methods must first be registered in the script system. You may place the registration code in a file with the same name as the component code file, with an added 'RTTI' suffix (frxBitBtnRTTI.pas in our case). See more about classes, registration, methods and properties in the FastScript script library documentation.

```
uses fs_iinterpreter, frxBitBtn, frxClassRTTI;

type
  TFunctions = class(TfsRTTIModule)
  public
    constructor Create(AScript: TfsScript); override;
  end;


constructor TFunctions.Create(AScript: TfsScript);
begin
  inherited Create(AScript);
  with AScript do
  begin
    { register class and then define its parent }
    AddClass(TfrxBitBtnControl, 'TfrxDialogControl');

    { if there are several common controls in your unit,
      they can be registered here;
      for example, AddClass(TfrxAnotherControl, 'TfrxDialogControl'); }
  end;
end;


initialization
  fsRTTIModules.Add(TFunctions);

end.
```

Writing Component Editors

All common control editors (opened from a control's context menu or by double-clicking) create blank OnClick event handlers by default. This behavior can be altered by writing a custom editor. Also, the custom editor can add extra items to the component's context menu.

The base class for all editors is “TfrxComponentEditor”, declared in the frxDsgnIntf file:

```
TfrxComponentEditor = class(TObject)
protected
  function AddItem(Caption: String; Tag: Integer;
                   Checked: Boolean = False): TMenuItem;
public
  function Edit: Boolean; virtual;
  function HasEditor: Boolean; virtual;
  function Execute(Tag: Integer; Checked: Boolean): Boolean; virtual;
  procedure GetMenuItems; virtual;
  property Component: TfrxComponent readonly;
  property Designer: TfrxCustomDesigner readonly;
end;
```

If your editor does not create its own items in the contextual menu you will need to override two methods, “Edit” and “HasEditor.” The first method performs essential actions (for example, displays the dialogue box) and returns “True” if the component's content was modified. The “HasEditor” method should return “True” if your component has an editor. If it either returns “False” or the method is not overridden the editor will not be opened. It should return “False” if your component does not have an editor and you wish to add items to the component's context menu.

If the editor adds items to the context menu you should override “GetMenuItems” (where you can create a menu using the AddItem function) and “Execute” (where you can create the actions initiated by selecting the items in the component menu).

Editor registration is performed using the procedure defined in the “frxDsgnIntf” file:

```
frxComponentEditors.Register(ComponentClass: TfrxComponentClass;
                             ComponentEditor: TfrxComponentEditorClass);
```

The first parameter is the class name of the component for which the editor is being created. The second parameter is the class name of the editor.

Let's look at a simple editor for our common control, which will display a window with our element name and also add "Enabled" and "Visible" items to control's context menu (which change the “Enabled” and “Visible” properties). FastReport requires that the Editor code is placed in a file having the same name as the file having the component’s code, with 'Editor' added as suffix (frxBitBtnEditor.pas in our case).

```
uses frxClass, frxDsgnIntf, frxBitBtn;

type
  TfrxBitBtnEditor = class(TfrxComponentEditor)
  public
    function Edit: Boolean; override;
    function HasEditor: Boolean; override;
    function Execute(Tag: Integer; Checked: Boolean): Boolean; override;
    procedure GetMenuItems; override;
  end;

function TfrxBitBtnEditor.Edit: Boolean;
var
  c: TfrxBitBtnControl;
begin
  Result := False;
  { Component property is edited component;
    in this case, it is  TfrxBitBtnControl }
  c := TfrxBitBtnControl(Component);
  ShowMessage('This is ' + c.Name);
end;

function TfrxBitBtnEditor.HasEditor: Boolean;
begin
  Result := True;
end;

function TfrxBitBtnEditor.Execute(Tag: Integer; Checked: Boolean):
Boolean;
var
  c: TfrxBitBtnControl;
begin
  Result := True;
  c := TfrxBitBtnControl(Component);
  if Tag = 1 then
    c.Enabled := Checked
  else if Tag = 2 then
    c.Visible := Checked;
end;

procedure TfrxBitBtnEditor.GetMenuItems;
var
  c: TfrxBitBtnControl;
begin
  c := TfrxBitBtnControl(Component);
  { AddItem method parameters: menu item name,
    its tag and Checked/Unchecked condition }
  AddItem('Enabled', 1, c.Enabled);
  AddItem('Visible', 2, c.Visible);
end;

initialization
  frxComponentEditors.Register(TfrxBitBtnControl, TfrxBitBtnEditor);

end.
```

Writing Property Editors

When you select a component in the designer its properties are displayed in the object inspector. You can create your own editor for any property. The “Font” property, for example, has an editor : if this property is selected a '...' button appears to the right of the line; open the standard "font properties" dialogue by clicking this button. Another example is the “Color” property. It opens the standard colors and color specimens names in a drop-down list.

“TfrxPropertyEditor” is the base class for all property editors and is declared in the “frxDsgnIntf” unit:

```
TfrxPropertyEditor = class(TObject)
protected
  procedure GetStrProc(const s: String);
  function GetFloatValue: Extended;
  function GetOrdValue: Integer;
  function GetStrValue: String;
  function GetVarValue: Variant;
  procedure SetFloatValue(Value: Extended);
  procedure SetOrdValue(Value: Integer);
  procedure SetStrValue(const Value: String);
  procedure SetVarValue(Value: Variant);
public
  constructor Create(Designer: TfrxCustomDesigner); virtual;
  destructor Destroy; override;
  function Edit: Boolean; virtual;
  function GetAttributes: TfrxPropertyAttributes; virtual;
  function GetExtraLBSize: Integer; virtual;
  function GetValue: String; virtual;
  procedure GetValues; virtual;
  procedure SetValue(const Value: String); virtual;
  procedure OnDrawLBItem(Control: TWinControl; Index: Integer;
                       ARect: TRect; State: TOwnerDrawState); virtual;
  procedure OnDrawItem(Canvas: TCanvas; ARect: TRect); virtual;
  property Component: TPersistent readonly;
  property frComponent: TfrxComponent readonly;
  property Designer: TfrxCustomDesigner readonly;
  property ItemHeight: Integer;
  property PropInfo: PPropInfo readonly;
  property Value: String;
  property Values: TStrings readonly;
end;
```

You also can inherit from any of the following classes, which provide some basic functionality for working with a property of the corresponding type:

```
TfrxIntegerProperty = class(TfrxPropertyEditor)
TfrxFloatProperty = class(TfrxPropertyEditor)
TfrxCharProperty = class(TfrxPropertyEditor)
TfrxStringProperty = class(TfrxPropertyEditor)
TfrxEnumProperty = class(TfrxPropertyEditor)
TfrxClassProperty = class(TfrxPropertyEditor)
TfrxComponentProperty = class(TfrxPropertyEditor)
```

Several properties are defined in the TfrxPropertyEditor class:

- Component	link to parent component (not to property itself!) to which the given property belongs
- *frComponent*	the same, but cast to TfrxComponent type (convenient in some cases)
- *Designer*	link to the report designer
- *ItemHeight* OnDrawXXX	item height, in which property is displayed; it can be useful in
- PropInfo	link to PPropInfo structure, which contains information about the edited property
- *Value*	property value displayed as a string
- Values	list of values; this property is to be filled by the “GetValue” method, if the “paValueList” attribute is defined (see below)

The following are system methods. They can be used to get or set the edited property value.

```
function GetFloatValue: Extended;
function GetOrdValue: Integer;
function GetStrValue: String;
function GetVarValue: Variant;
procedure SetFloatValue(Value: Extended);
procedure SetOrdValue(Value: Integer);
procedure SetStrValue(const Value: String);
procedure SetVarValue(Value: Variant);
```

You should use the methods appropriate to the property type. So, use “GetOrdValue” and “SetOrdValue” methods if the property is of “Integer” type, etc.. These methods are also used for working with properties of “TObject" type, since these properties contain a 32-bit object address. In these cases, just use a cast like: MyFont := TFont(GetOrdValue).

To create your own editor, inherit from the base class and override some methods declared in the public section (which methods is dependent on the property type and functionality you wish to implement). One of the methods that will need to be overridden is “GetAttributes” which returns a set of property attributes. Attributes are defined in the following way:

```
TfrxPropertyAttribute = (paValueList, paSortList, paDialog, paMultiSelect,
                         paSubProperties, paReadOnly, paOwnerDraw);
TfrxPropertyAttributes = set of TfrxPropertyAttribute;
```

Attribute meanings are:

- *paValueList*	property represents a drop-down list of values (for example the “Color” property); if present then the “GetValues” method should be overridden
- *paSortList*	sorts list values; it is used together with paValueList
- *paDialog*	property has an editor; if present a '...' button is displayed to the right of the line; Edit method is called by clicking on it
- *paMultiSelect*	allows more than one object to be selected at the same time; some properties (such as “Name”, etc) do not have this attribute
- *paSubProperties*	property is object of TPersistent type and has its own properties, which should also be displayed (for example the “Font” property)
- *paReadOnly*	value cannot be changed in the editor; some properties, of “Class” or “Set” types, have this attribute
- *paOwnerDraw*	property value is drawn by the “OnDrawItem” method; if the “paValueList” attribute is present then a drop-down list is drawn by the OnDrawLBItem method

The Edit method is called in two ways: either by selecting a property and double-clicking its value or (if property has paDialog attribute) by clicking the '...' button. This method should return “True” if property value was modified.

The “GetValue” method should return the property value as a string (it will be displayed in the object inspector). If you inherit from the TfrxPropertyEditor base class this method must be overridden.

The “SetValue” method is to set the property value, transferred as a string. If you inherit from the TfrxPropertyEditor base class this method must be overridden.

The “GetValues” method should be overridden if the “paValueList” attribute is set. This method should fill the “Values” property with values.

The following three methods allow manual drawing of the property value (the Color property editor works in the same way). These methods are called if the “paOwnerDraw” attribute is set.

The “OnDrawItem” method is called when the property value is drawn in the object inspector (when the property is not selected; otherwise its value is simply displayed in the editing line). For example, the Color property editor draws a rectangle to the left of property value, filled with the corresponding color.

The “GetExtraLBSize” method is called if the “paValueList” attribute is set. This method returns the number of pixels by which to adjust the “Drop-Down List” width to display the list. By default this method returns the value corresponding to the cell height. If you need to draw picture, with width larger than its height, the given method should be overridden.

The “OnDrawLBItem” method is called when drawing a string in a drop-down list if the paValueList attribute is set. This method is actually the TListBox.OnDrawItem event handler and has the same set of parameters.

Property editor registration is performed by the procedure defined in the frxDsgnIntf file:

```
procedure frxPropertyEditors.Register(PropertyType: PTypeInfo;
                                      ComponentClass: TClass;
                                      const PropertyName: String;
                                      EditorClass:
TfrxPropertyEditorClass);
```

- *PropertyType*	information about the property type, returned by the “TypeInfo” system function, for example TypeInfo(String)
- *ComponentClass*	component name, with property you want to edit (may be nil)
- *PropertyName*	name of property you want to edit (may be a blank string)
- *EditorClass*	property editor name

Only the “PropertyType” parameter needs to be specified. The “ComponentClass” and/or “PropertyName” parameters may be blank. This allows the editor to be registered either to any property of PropertyType type, to any property of concrete ComponentClass components and its descendants, or to a PropertyName specific property of a specific component (or to any component if the ComponentClass parameter is blank).

Let's look at three property editor examples. FastReport requires the Editor code to be placed in a file having the same name as the file having the component code, with 'Editor' suffixed to it.

```
--------------------------------------------------
```

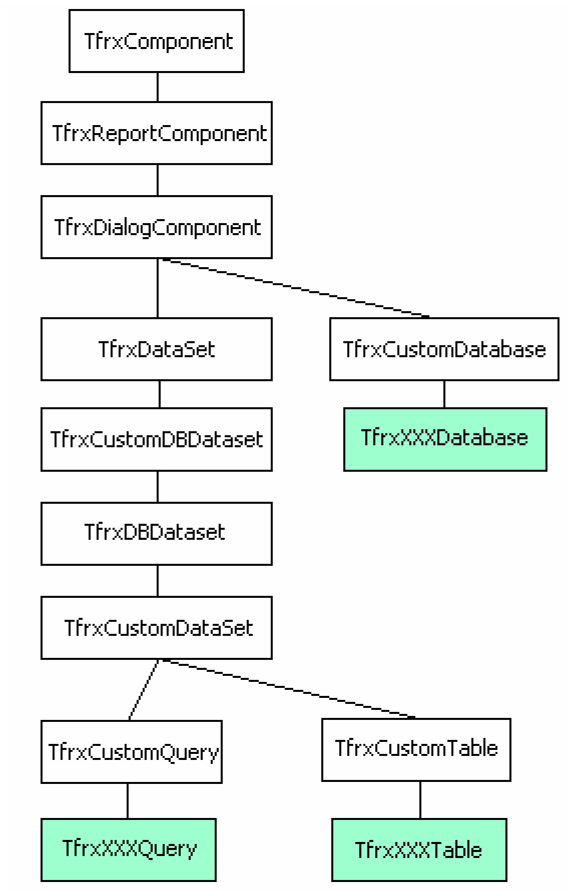
```
{ TFont property editor displays editor button('...') }
{ inherit from ClassProperty }
type
  TfrxFontProperty = class(TfrxClassProperty)
  public
    function Edit: Boolean; override;
    function GetAttributes: TfrxPropertyAttributes; override;
  end;

function TfrxFontProperty.GetAttributes: TfrxPropertyAttributes;
begin
  { property has nested properties and  editor;
    it cannot be edited manually }
  Result := [paMultiSelect, paDialog, paSubProperties, paReadOnly];
end;

function TfrxFontProperty.Edit: Boolean;
var
  FontDialog: TFontDialog;
begin
  { create standard dialogue }
  FontDialog := TFontDialog.Create(Application);
  try
    { take  property value }
    FontDialog.Font := TFont(GetOrdValue);
    FontDialog.Options := FontDialog.Options + [fdForceFontExist];
    { display dialogue }
    Result := FontDialog.Execute;
    { bind new value }
    if Result then
      SetOrdValue(Integer(FontDialog.Font));
  finally
    FontDialog.Free;
  end;
end;

{ registration }
frxPropertyEditors.Register(TypeInfo(TFont), nil, '', TfrxFontProperty);

--------------------------------------------------

{ TFont.Name property editor displays a drop-down list
  of available fonts;
  inherit from StringProperty, as property is of string type }
type
  TfrxFontNameProperty = class(TfrxStringProperty)
  public
    function GetAttributes: TfrxPropertyAttributes; override;
    procedure GetValues; override;
  end;

function TfrxFontNameProperty.GetAttributes: TfrxPropertyAttributes;
begin
  Result := [paMultiSelect, paValueList];
end;

procedure TfrxFontNameProperty.GetValues;
begin
  Values.Assign(Screen.Fonts);
end;
```

```
{ registration }
frxPropertyEditors.Register(TypeInfo(String), TFont,
                            'Name', TfrxFontNameProperty);


--------------------------------------------------

{ TPen.Style property editor displays a picture,
  which is an example of the selected style }
type
  TfrxPenStyleProperty = class(TfrxEnumProperty)
  public
    function GetAttributes: TfrxPropertyAttributes; override;
    function GetExtraLBSize: Integer; override;
    procedure OnDrawLBItem(Control: TWinControl; Index: Integer;
      ARect: TRect; State: TOwnerDrawState); override;
    procedure OnDrawItem(Canvas: TCanvas; ARect: TRect); override;
  end;

function TfrxPenStyleProperty.GetAttributes: TfrxPropertyAttributes;
begin
  Result := [paMultiSelect, paValueList, paOwnerDraw];
end;

{ method draws thick horizontal line with selected style }
procedure HLine(Canvas: TCanvas; X, Y, DX: Integer);
var
  i: Integer;
begin
  with Canvas do
  begin
    Pen.Color := clBlack;
    for i := 0 to 1 do
    begin
      MoveTo(X, Y - 1 + i);
      LineTo(X + DX, Y - 1 + i);
    end;
  end;
end;

{ drawing drop-down list }
procedure TfrxPenStyleProperty.OnDrawLBItem
               (Control: TWinControl; Index: Integer;
                ARect: TRect; State: TOwnerDrawState);
begin
  with TListBox(Control), TListBox(Control).Canvas do
  begin
    FillRect(ARect);
    TextOut(ARect.Left + 40, ARect.Top + 1, TListBox(Control).Items
[Index]);

    Pen.Color := clGray;
    Brush.Color := clWhite;
    Rectangle(ARect.Left + 2, ARect.Top + 2,
              ARect.Left + 36, ARect.Bottom - 2);

    Pen.Style := TPenStyle(Index);
    HLine(TListBox(Control).Canvas, ARect.Left + 3,
          ARect.Top + (ARect.Bottom - ARect.Top) div 2, 32);
    Pen.Style := psSolid;
  end;
end;

{ drawing property value }
procedure TfrxPenStyleProperty.OnDrawItem(Canvas: TCanvas; ARect: TRect);
begin
  with Canvas do
  begin
    TextOut(ARect.Left + 38, ARect.Top, Value);

    Pen.Color := clGray;
    Brush.Color := clWhite;
    Rectangle(ARect.Left, ARect.Top + 1,
              ARect.Left + 34, ARect.Bottom - 4);

    Pen.Color := clBlack;
    Pen.Style := TPenStyle(GetOrdValue);
    HLine(Canvas, ARect.Left + 1,
          ARect.Top + (ARect.Bottom - ARect.Top) div 2 - 1, 32);
    Pen.Style := psSolid;
  end;
end;

{ return picture width }
function TfrxPenStyleProperty.GetExtraLBSize: Integer;
begin
  Result := 36;
end;

{ registration }
frxPropertyEditors.Register(TypeInfo(TPenStyle), TPen,
                            'Style', TfrxPenStyleProperty);
```

Writing Custom DB Engines

FastReport can build reports not only with data sourced from a Delphi application but also from data sources (connections to DBs, queries) created within the report itself. FastReport comes with engines for ADO, BDE, IBX, DBX and FIB. You can create your own engine and then connect it to FastReport.

The illustration below shows the class hierarchy required for creating DB engines. New engine components are highlighted in green.



A standard set of DB engine components includes Database, Table and Query. You can create all of these components or just some of them (for example many DBs have no component of the Table type). You can also create components which are not included in the standard set (for example a StoredProc).

Let's look at the base classes in detail.

“TfrxDialogComponent” is the base class for all non-visual components that can be placed on a FastReport design dialogue form. It has no any important properties or methods defined within it.

“TfrxCustomDatabase” class is the base class for DB components of “Database” type.

```
TfrxCustomDatabase = class(TfrxDialogComponent)
```

```
protected
  procedure SetConnected(Value: Boolean); virtual;
  procedure SetDatabaseName(const Value: String); virtual;
  procedure SetLoginPrompt(Value: Boolean); virtual;
  procedure SetParams(Value: TStrings); virtual;
  function GetConnected: Boolean; virtual;
  function GetDatabaseName: String; virtual;
  function GetLoginPrompt: Boolean; virtual;
  function GetParams: TStrings; virtual;
public
  procedure SetLogin(const Login, Password: String); virtual;
  property Connected: Boolean read GetConnected
                              write SetConnected default False;
  property DatabaseName: String read GetDatabaseName
                                write SetDatabaseName;
  property LoginPrompt: Boolean read GetLoginPrompt
                                write SetLoginPrompt default True;
  property Params: TStrings read GetParams
                            write SetParams;
end;
```

The following properties are defined in this class:

- *Connected*	whether DB connection is active
- *DatabaseName*	database name
- *LoginPrompt*	whether to ask for login when connecting to DB
- *Params*	connection parameters

Inherit from this class to create a component of TfrxXXXDatabase type. Once created all virtual methods must be overridden and any required properties placed in the published section. Also add any properties specific for your component.

The **“TfrxDataset”**, **“TfrxCustomDBDataset”** and **“TfrxDBDataset”** classes provide data access functions. The FastReport core uses these components for navigation and referencing data fields. As such they are part of the common hierarchy and are of no interest to us.

“TfrxCustomDataSet” is a base class for DB components derived from TDataSet. Components inheriting from this class are “Query”, “Table” and “StoredProc” clones. Actually this class is a wrapper for TDataSet.

```
TfrxCustomDataset = class(TfrxDBDataSet)
protected
  procedure SetMaster(const Value: TDataSource); virtual;
  procedure SetMasterFields(const Value: String); virtual;
public
  property DataSet: TDataSet;
  property Fields: TFields readonly;
  property MasterFields: String;
  property Active: Boolean;
published
  property Filter: String;
  property Filtered: Boolean;
  property Master: TfrxDBDataSet;
end;
```

The following properties are defined in this class:

- *DataSet*	a link to the enclosed object of “TdataSet” type
- *Fields*	a link to DataSet.Fields
- *Active*	whether the DataSet is active
- *Filter*	expression for filtering
- *Filtered*	whether filtering is active
- *Master*	a link to the master dataset in a master-detail relationship
- *MasterFields*	list of fields like 'field1=field2'; used for master-detail relationships

“TfrxCustomTable” is the base class for DB components of Table type. This class is a wrapper for the TTable class.

```
TfrxCustomTable = class(TfrxCustomDataset)
protected
  function GetIndexFieldNames: String; virtual;
  function GetIndexName: String; virtual;
  function GetTableName: String; virtual;
  procedure SetIndexFieldNames(const Value: String); virtual;
  procedure SetIndexName(const Value: String); virtual;
  procedure SetTableName(const Value: String); virtual;
published
  property MasterFields;
  property TableName: String read GetTableName write SetTableName;
  property IndexName: String read GetIndexName write SetIndexName;
  property IndexFieldNames: String read GetIndexFieldNames
                                   write SetIndexFieldNames;
end;
```

The following properties are defined in the class:

- *TableName*	table name
- *IndexName*	index name
- *IndexFieldNames*	index field names

Components of Table type inherit from this class. When creating a descendant of this class you should add some missing properties like Database. Also, the virtual methods of the TfrxCustomDataset and TfrxCustomTable classes must be overridden.

“TfrxCustomQuery” is the base class for DB components of “Query” type. This class is a wrapper for the TQuery class.

```
TfrxCustomQuery = class(TfrxCustomDataset)
protected
  procedure SetSQL(Value: TStrings); virtual; abstract;
  function GetSQL: TStrings; virtual; abstract;
public
  procedure UpdateParams; virtual; abstract;
published
  property Params: TfrxParams;
  property SQL: TStrings;
end;
```

The “SQL” and “Params” properties (found in all Query components) are declared in this class. Since Query components can implement parameters in different ways, for example as TParams or TParameters, the “Params” property is of “TfrxParams” type, which is a wrapper for all parameter types.

The following methods are declared in this class:

- *SetSQL*	sets “SQL” property of “Query” type
- *GetSQL*	gets “SQL” property of “Query” type
- *UpdateParams*	copies parameter values into component of Query type; if the Query component parameters are of TParams type then copying is by means of the frxParamsToTParams standard procedure

Let's demonstrate the creation of a DB engine using an IBX example. The full text for the engine can be found in the SOURCE\IBX folder. Below are some extracts from the source text, with added comments.

The IBX components around which we will build the wrapper are TIBDatabase, TIBTable and TIBQuery. Our corresponding components will be named “TfrxIBXDatabase”, “TfrxIBXTable” and “TfrxIBXQuery”.

“TfrxIBXComponents” is another component that we should create; it will be placed on the FastReport component palette when registering the engine in the Delphi environment. As soon as this component is used in a project Delphi will automatically add a link to our engine unit in the “Uses” list. There is one more task to complete for this component, to define the “DefaultDatabase” property, which references an existing connection to a DB. By default all TfrxIBXTable and TfrxIBXQuery components will use this connection. The TfrxIBXComponents component must inherit from from the TfrxDBComponents class:

```
TfrxDBComponents = class(TComponent)
public
  function GetDescription: String; virtual; abstract;
end;
```

A description should be returned by one function only, for example ”IBX Components”. A “TfrxIBXComponents” component is declared as follows:

```
type
  TfrxIBXComponents = class(TfrxDBComponents)
  private
    FDefaultDatabase: TIBDatabase;
    FOldComponents: TfrxIBXComponents;
  public
    constructor Create(AOwner: TComponent); override;
    destructor Destroy; override;
    function GetDescription: String; override;
  published
    property DefaultDatabase: TIBDatabase read FDefaultDatabase
                                          write FDefaultDatabase;
  end;

var
  IBXComponents: TfrxIBXComponents;

constructor TfrxIBXComponents.Create(AOwner: TComponent);
begin
  inherited;
  FOldComponents := IBXComponents;
  IBXComponents := Self;
end;

destructor TfrxIBXComponents.Destroy;
begin
```

```
    if IBXComponents = Self then
      IBXComponents := FOldComponents;
    inherited;
  end;

  function TfrxIBXComponents.GetDescription: String;
  begin
    Result := 'IBX';
  end;
```

We declare an IBXComponents global variable which will reference a copy of the TfrxIBXComponents component. If you put the component into the project several times (which is pointless) you will nevertheless be able to save a link to the previous component and restore it after deleting the component.

A link to a DB connection that already exists in the project can be set in the “DefaultDatabase” property. The way we will write the TfrxIBXTable and TfrxIBXQuery components will allow them to use this connection by default (actually, this is the purpose of the IBXComponents global variable).

Here is the TfrxIBXDatabase component. It is a wrapper for the TIBDatabase class.

```
  TfrxIBXDatabase = class(TfrxCustomDatabase)
  private
    FDatabase: TIBDatabase;
    FTransaction: TIBTransaction;
    function GetSQLDialect: Integer;
    procedure SetSQLDialect(const Value: Integer);
  protected
    procedure SetConnected(Value: Boolean); override;
    procedure SetDatabaseName(const Value: String); override;
    procedure SetLoginPrompt(Value: Boolean); override;
    procedure SetParams(Value: TStrings); override;
    function GetConnected: Boolean; override;
    function GetDatabaseName: String; override;
    function GetLoginPrompt: Boolean; override;
    function GetParams: TStrings; override;
  public
    constructor Create(AOwner: TComponent); override;
    destructor Destroy; override;
    class function GetDescription: String; override;
    procedure SetLogin(const Login, Password: String); override;
    property Database: TIBDatabase read FDatabase;
  published
    { list TIBDatabase properties.
      Note – some properties already exist in base class }
    property DatabaseName;
    property LoginPrompt;
    property Params;
    property SQLDialect: Integer read GetSQLDialect write SetSQLDialect;
    { Connected property should be placed last! }
    property Connected;
  end;

  constructor TfrxIBXDatabase.Create(AOwner: TComponent);
  begin
    inherited;
    { create component – connection }
    FDatabase := TIBDatabase.Create(nil);
    { create component - transaction (specific to IBX) }
    FTransaction := TIBTransaction.Create(nil);
    FDatabase.DefaultTransaction := FTransaction;
    { don't forget this line! }
    Component := FDatabase;
  end;

  destructor TfrxIBXDatabase.Destroy;
  begin
    { delete  transaction }
    FTransaction.Free;
    { connection will be deleted automatically in parent class }
    inherited;
  end;

  { component description will be displayed next to icon
    in objects toolbar }
  class function TfrxIBXDatabase.GetDescription: String;
  begin
    Result := 'IBX Database';
  end;

  { redirect component properties to cover properties and vice versa }
  function TfrxIBXDatabase.GetConnected: Boolean;
  begin
    Result := FDatabase.Connected;
  end;

  function TfrxIBXDatabase.GetDatabaseName: String;
  begin
    Result := FDatabase.DatabaseName;
  end;

  function TfrxIBXDatabase.GetLoginPrompt: Boolean;
  begin
    Result := FDatabase.LoginPrompt;
  end;

  function TfrxIBXDatabase.GetParams: TStrings;
  begin
    Result := FDatabase.Params;
  end;

  function TfrxIBXDatabase.GetSQLDialect: Integer;
  begin
    Result := FDatabase.SQLDialect;
  end;

  procedure TfrxIBXDatabase.SetConnected(Value: Boolean);
  begin
    FDatabase.Connected := Value;
    FTransaction.Active := Value;
  end;

  procedure TfrxIBXDatabase.SetDatabaseName(const Value: String);
  begin
    FDatabase.DatabaseName := Value;
  end;

  procedure TfrxIBXDatabase.SetLoginPrompt(Value: Boolean);
```

```
begin
  FDatabase.LoginPrompt := Value;
end;

procedure TfrxIBXDatabase.SetParams(Value: TStrings);
begin
  FDatabase.Params := Value;
end;

procedure TfrxIBXDatabase.SetSQLDialect(const Value: Integer);
begin
  FDatabase.SQLDialect := Value;
end;

{ this method is used by DB connection wizard }
procedure TfrxIBXDatabase.SetLogin(const Login, Password: String);
begin
  Params.Text := 'user_name=' + Login + #13#10 + 'password=' + Password;
end;
```

As you can see, this is not that complicated. We created FDatabase : “TIBDatabase” object and then defined the properties we want the designer to show. “Get” and “Set” methods were written for each property.

The next class is **“TfrxIBXTable”**. As mentioned above, it inherits from the TfrxCustomDataSet standard class. All the basic functionality (operating with a list of fields, master-detail and basic properties) is already implemented in the base class. We only need to declare properties that are specific to this component.

```
TfrxIBXTable = class(TfrxCustomTable)
private
  FDatabase: TfrxIBXDatabase;
  FTable: TIBTable;
  procedure SetDatabase(const Value: TfrxIBXDatabase);
protected
  procedure Notification(AComponent: TComponent; Operation: TOperation);
override;
  procedure SetMaster(const Value: TDataSource); override;
  procedure SetMasterFields(const Value: String); override;
  procedure SetIndexFieldNames(const Value: String); override;
  procedure SetIndexName(const Value: String); override;
  procedure SetTableName(const Value: String); override;
  function GetIndexFieldNames: String; override;
  function GetIndexName: String; override;
  function GetTableName: String; override;
public
  constructor Create(AOwner: TComponent); override;
  constructor DesignCreate(AOwner: TComponent; Flags: Word); override;
  class function GetDescription: String; override;
  procedure BeforeStartReport; override;
  property Table: TIBTable read FTable;
published
  property Database: TfrxIBXDatabase read FDatabase write SetDatabase;
end;

constructor TfrxIBXTable.Create(AOwner: TComponent);
begin
  { create component – table }
```

```
    FTable := TIBTable.Create(nil);
    { assign link to DataSet property from basic class
      – don't forget this string! }
    DataSet := FTable;
    { assign link to connection to DB by default }
    SetDatabase(nil);
    { after that basic constructor may be called in}
    inherited;
  end;

  { this constructor is called at the moment of adding components to report;
    it connects table to TfrxIBXDatabase component automatically,
    if it is already present }
  constructor TfrxIBXTable.DesignCreate(AOwner: TComponent; Flags: Word);
  var
    i: Integer;
    l: TList;
  begin
    inherited;
    l := Report.AllObjects;
    for i := 0 to l.Count - 1 do
      if TObject(l[i]) is TfrxIBXDatabase then
      begin
        SetDatabase(TfrxIBXDatabase(l[i]));
        break;
      end;
  end;

  class function TfrxIBXTable.GetDescription: String;
  begin
    Result := 'IBX Table';
  end;

  { trace TfrxIBXDatabase component deletion; we address this component
    in FDatabase property; otherwise can generate an error }
  procedure TfrxIBXTable.Notification(AComponent: TComponent; Operation:
  TOperation);
  begin
    inherited;
    if (Operation = opRemove) and (AComponent = FDatabase) then
      SetDatabase(nil);
  end;

  procedure TfrxIBXTable.SetDatabase(const Value: TfrxIBXDatabase);
  begin
    { Database property of TfrxIBXDatabase type, not of TIBDatabase type! }
   FDatabase := Value;
    { if value <> nil, connect table to selected component }
    if Value <> nil then
      FTable.Database := Value.Database
    { otherwise try to connect to DB by default,
      defined in TfrxIBXComponents component }
    else if IBXComponents <> nil then
      FTable.Database := IBXComponents.DefaultDatabase
    { if there were no TfrxIBXComponents for some reason, reset to nil }
    else
      FTable.Database := nil;
    { if connection was successful DBConnected flag should be put }
    DBConnected := FTable.Database <> nil;
  end;
```

```
function TfrxIBXTable.GetIndexFieldNames: String;
begin
  Result := FTable.IndexFieldNames;
end;

function TfrxIBXTable.GetIndexName: String;
begin
  Result := FTable.IndexName;
end;

function TfrxIBXTable.GetTableName: String;
begin
  Result := FTable.TableName;
end;

procedure TfrxIBXTable.SetIndexFieldNames(const Value: String);
begin
  FTable.IndexFieldNames := Value;
end;

procedure TfrxIBXTable.SetIndexName(const Value: String);
begin
  FTable.IndexName := Value;
end;

procedure TfrxIBXTable.SetTableName(const Value: String);
begin
  FTable.TableName := Value;
end;

procedure TfrxIBXTable.SetMaster(const Value: TDataSource);
begin
  FTable.MasterSource := Value;
end;

procedure TfrxIBXTable.SetMasterFields(const Value: String);
begin
  FTable.MasterFields := Value;
  FTable.IndexFieldNames := Value;
end;

{ we need to implement this method in some cases }
procedure TfrxIBXTable.BeforeStartReport;
begin
  SetDatabase(FDatabase);
end;
```

Finally, let’s look at the last component, **“TfrxIBXQuery”**. It inherits from the TfrxCustomQuery base class, in which the required properties are already declared. We only need to declare the Database property and override the SetMaster method. The implementation of the other methods is similar to the TfrxIBXTable component.

```
TfrxIBXQuery = class(TfrxCustomQuery)
private
  FDatabase: TfrxIBXDatabase;
  FQuery: TIBQuery;
  procedure SetDatabase(const Value: TfrxIBXDatabase);
protected
```

```
  procedure Notification(AComponent: TComponent; Operation: TOperation);
override;
  procedure SetMaster(const Value: TDataSource); override;
  procedure SetSQL(Value: TStrings); override;
  function GetSQL: TStrings; override;
public
  constructor Create(AOwner: TComponent); override;
  constructor DesignCreate(AOwner: TComponent; Flags: Word); override;
  class function GetDescription: String; override;
  procedure BeforeStartReport; override;
  procedure UpdateParams; override;
  property Query: TIBQuery read FQuery;
published
  property Database: TfrxIBXDatabase read FDatabase write SetDatabase;
end;


constructor TfrxIBXQuery.Create(AOwner: TComponent);
begin
  { create component – query }
  FQuery := TIBQuery.Create(nil);
  { assign link to it to DataSet property from base class
    – don't forget this line! }
  Dataset := FQuery;
  { assign link to connection to DB by default }
  SetDatabase(nil);
  { after that base constructor may be called }
  inherited;
end;

constructor TfrxIBXQuery.DesignCreate(AOwner: TComponent; Flags: Word);
var
  i: Integer;
  l: TList;
begin
  inherited;
  l := Report.AllObjects;
  for i := 0 to l.Count - 1 do
    if TObject(l[i]) is TfrxIBXDatabase then
    begin
      SetDatabase(TfrxIBXDatabase(l[i]));
      break;
    end;
end;

class function TfrxIBXQuery.GetDescription: String;
begin
  Result := 'IBX Query';
end;

procedure TfrxIBXQuery.Notification(AComponent: TComponent;
                                     Operation: TOperation);
begin
  inherited;
  if (Operation = opRemove) and (AComponent = FDatabase) then
    SetDatabase(nil);
end;

procedure TfrxIBXQuery.SetDatabase(const Value: TfrxIBXDatabase);
begin
  FDatabase := Value;
```

```
    if Value <> nil then
      FQuery.Database := Value.Database
    else if IBXComponents <> nil then
      FQuery.Database := IBXComponents.DefaultDatabase
    else
      FQuery.Database := nil;
    DBConnected := FQuery.Database <> nil;
  end;

  procedure TfrxIBXQuery.SetMaster(const Value: TDataSource);
  begin
    FQuery.DataSource := Value;
  end;

  function TfrxIBXQuery.GetSQL: TStrings;
  begin
    Result := FQuery.SQL;
  end;

  procedure TfrxIBXQuery.SetSQL(Value: TStrings);
  begin
    FQuery.SQL := Value;
  end;

  procedure TfrxIBXQuery.UpdateParams;
  begin
    { in this method it is sufficient to assign values
      from Params to FQuery.Params }
    { this is performed via standard procedure }
    frxParamsToTParams(Self, FQuery.Params);
  end;

  procedure TfrxIBXQuery.BeforeStartReport;
  begin
    SetDatabase(FDatabase);
  end;
```

Registration of all engine components is performed in the “Initialization” section.

```
  initialization
    { use standard pictures indexes 37,38,39 instead of pictures}
    frxObjects.RegisterObject1(TfrxIBXDataBase, nil, '', '', 0, 37);
    frxObjects.RegisterObject1(TfrxIBXTable, nil, '', '', 0, 38);
    frxObjects.RegisterObject1(TfrxIBXQuery, nil, '', '', 0, 39);

  finalization
    frxObjects.Unregister(TfrxIBXDataBase);
    frxObjects.Unregister(TfrxIBXTable);
    frxObjects.Unregister(TfrxIBXQuery);

  end.
```

This is enough to use the engine in reports. There are two more things left at this stage: to register the engine classes in the script system so that they can be referred to in the script, and to register several property editors (for example TfrxIBXTable.TableName) to make working with the component easier.

It is better to store the engine registration code in a separate file with a RTTI suffix. See more about class registration in the script system in the appropriate chapter. Here is an example of a file:

```
unit frxIBXRTTI;

interface

{$I frx.inc}

implementation

uses
  Windows, Classes, fs_iinterpreter, frxIBXComponents
{$IFDEF Delphi6}
, Variants
{$ENDIF};

type
  TFunctions = class(TfsRTTIModule)
  public
    constructor Create(AScript: TfsScript); override;
  end;


{ TFunctions }

constructor TFunctions.Create;
begin
  inherited Create(AScript);
  with AScript do
  begin
    AddClass(TfrxIBXDatabase, 'TfrxComponent');
    AddClass(TfrxIBXTable, 'TfrxCustomDataset');
    AddClass(TfrxIBXQuery, 'TfrxCustomQuery');
  end;
end;

initialization
  fsRTTIModules.Add(TFunctions);

end.
```

It is also recommended that property editor code is put in a separate file with an 'Editor' suffix. In our case, editors for the TfrxIBXDatabase.DatabaseName, TfrxIBXTable.IndexName and TfrxIBXTable.TableName properties were required. See more about writing property editors in the appropriate chapter. Below is an example of a file:

```
unit frxIBXEditor;

interface

{$I frx.inc}

implementation

uses
  Windows, Classes, SysUtils, Forms, Dialogs, frxIBXComponents,
frxCustomDB,
  frxDsgnIntf, frxRes, IBDatabase, IBTable
{$IFDEF Delphi6}
, Variants
{$ENDIF};
```

```
type
  TfrxDatabaseNameProperty = class(TfrxStringProperty)
  public
    function GetAttributes: TfrxPropertyAttributes; override;
    function Edit: Boolean; override;
  end;

  TfrxTableNameProperty = class(TfrxStringProperty)
  public
    function GetAttributes: TfrxPropertyAttributes; override;
    procedure GetValues; override;
  end;

  TfrxIndexNameProperty = class(TfrxStringProperty)
  public
    function GetAttributes: TfrxPropertyAttributes; override;
    procedure GetValues; override;
  end;


{ TfrxDatabaseNameProperty }

function TfrxDatabaseNameProperty.GetAttributes: TfrxPropertyAttributes;
begin
  { this property possesses an editor }
  Result := [paDialog];
end;

function TfrxDatabaseNameProperty.Edit: Boolean;
var
  SaveConnected: Bool;
  db: TIBDatabase;
begin
  { get link to TfrxIBXDatabase.Database }
  db := TfrxIBXDatabase(Component).Database;
  { create standard OpenDialog }
  with TOpenDialog.Create(nil) do
  begin
    InitialDir := GetCurrentDir;
    { we are interested in *.gdb files }
    Filter := frxResources.Get('ftDB') + ' (*.gdb)|*.gdb|'
              + frxResources.Get('ftAllFiles') + ' (*.*)|*.*';
    Result := Execute;
    if Result then
    begin
      SaveConnected := db.Connected;
      db.Connected := False;
      { if dialogue is completed successfully, assign new DB name }
      db.DatabaseName := FileName;
      db.Connected := SaveConnected;
    end;
    Free;
  end;
end;


{ TfrxTableNameProperty }

function TfrxTableNameProperty.GetAttributes: TfrxPropertyAttributes;
begin
  {  property represents list of values }
  Result := [paMultiSelect, paValueList];
end;

procedure TfrxTableNameProperty.GetValues;
var
  t: TIBTable;
begin
  inherited;
  { get link to TIBTable component }
  t := TfrxIBXTable(Component).Table;
  { fill list of tables available }
  if t.Database <> nil then
    t.DataBase.GetTableNames(Values, False);
end;


{ TfrxIndexProperty }

function TfrxIndexNameProperty.GetAttributes: TfrxPropertyAttributes;
begin
  { property represents list of values }
  Result := [paMultiSelect, paValueList];
end;

procedure TfrxIndexNameProperty.GetValues;
var
  i: Integer;
begin
  inherited;
  try
    { get link to TIBTable component }
    with TfrxIBXTable(Component).Table do
      if (TableName <> '') and (IndexDefs <> nil) then
      begin
        { update indexes }
        IndexDefs.Update;
        { fill list of indexes available }
        for i := 0 to IndexDefs.Count - 1 do
          if IndexDefs[i].Name <> '' then
            Values.Add(IndexDefs[i].Name);
      end;
  except
  end;
end;


initialization
  frxPropertyEditors.Register(TypeInfo(String), TfrxIBXDataBase,
                              'DatabaseName', TfrxDataBaseNameProperty);
  frxPropertyEditors.Register(TypeInfo(String), TfrxIBXTable,
                              'TableName', TfrxTableNameProperty);
  frxPropertyEditors.Register(TypeInfo(String), TfrxIBXTable,
                              'IndexName', TfrxIndexNameProperty);

end.
```

Using Custom Functions in a Report

FastReport has a large number of built-in standard functions for use in report designs. FastReport also allows custom functions to be written and used. Functions are added using the “FastScript” library interface, which is included in FastReport (to learn more about FastScript refer to it’s library manual).

Let's look at how procedures and/or functions can be added to FastReport. The number and types of parameters vary from function to function. Parameters of “Set” and “Record" type are not supported by FastScript, so they must be implemented using simpler types, for instance a TRect can be passed as four integers : X0, Y0, X1, Y1. There is more about the use of functions with various parameters in the FastScript documentation.

In the Delphi form declare the function or procedure and its code.

```
function TForm1.MyFunc(s: String; i: Integer): Boolean;
begin
// required logic
end;

procedure TForm1.MyProc(s: String);
begin
// required logic
end;
```

Create the “onUser” function handler for the report component.

```
function TForm1.frxReport1UserFunction(const MethodName: String;
                                       var Params: Variant): Variant;
begin
  if MethodName = 'MYFUNC' then
    Result := MyFunc(Params[0], Params[1])
  else if MethodName = 'MYPROC' then
    MyProc(Params[0]);
end;
```

Use the report component’s add method to add it to the function list (usually in the “onCreate” or “onShow” event of the Delphi form).

```
frxReport1.AddFunction('function MyFunc(s: String; i: Integer):Boolean');
frxReport1.AddFunction('procedure MyProc(s: String)');
```

The added function can now be used in a report script and can be referenced by objects of the “TfrxMemoView” type. The function is also displayed on the "Data tree" functions tab. On this tab functions are divided into categories and when selected a hint about the function appears in the bottom pane of the tab.

Modify the code sample above to register functions in separate categories, and to display descriptive hints:

```
frxReport1.AddFunction('function MyFunc(s: String; i: Integer): Boolean',
                       'My functions',
                       ' MyFunc function always returns True');
frxReport1.AddFunction('procedure MyProc(s: String)',
                       'My functions',
                       ' MyProc procedure does not do anything');
```

The added functions will appear under the category “My functions”.

To register functions in an existing categories use one of the following category names:

- *'ctString'*	string function
- *'ctDate'*	date/time functions
- *'ctConv'*	conversion functions
- *'ctFormat'*	formatting
- *'ctMath'*	mathematical functions
- *'ctOther'*	other functions

If the category name is left blank the function is placed under the functions tree root. To add a large number of functions it is recommended that all logic is placed in a separate library unit. Here is an example:

```
unit myfunctions;

interface

implementation

uses SysUtils, Classes, fs_iinterpreter;
           // you can also add a reference to any other external library
here

type
  TFunctions = class(TfsRTTIModule)
  private
    function CallMethod(Instance: TObject; ClassType: TClass;
                        const MethodName: String; var Params: Variant):
Variant;
  public
    constructor Create(AScript: TfsScript); override;
  end;


function MyFunc(s: String; i: Integer): Boolean;
begin
// required logic
end;

procedure MyProc(s: String);
begin
// required logic
end;


{ TFunctions }

constructor TFunctions.Create;
begin
  inherited Create(AScript);
  with AScript do
    AddMethod('function MyFunc(s: String; i: Integer): Boolean',
CallMethod,
              'My functions', ' MyFunc function always returns True');
    AddMethod('procedure MyProc(s: String)', CallMethod,
              'My functions', ' MyProc procedure does not do anything'');
  end;
```

```
end;

function TFunctions.CallMethod(Instance: TObject; ClassType: TClass;
                               const MethodName: String;
                               var Params: Variant): Variant;
begin
  if MethodName = 'MYFUNC' then
    Result := MyFunc(Params[0], Params[1])
  else if MethodName = 'MYPROC' then
    MyProc(Params[0]);
end;

initialization
  fsRTTIModules.Add(TFunctions);

end.
```

Save the file with a .pas extension then add a reference to it in the “uses” clause of your Delphi project’s form. All your custom functions will then be available for use in any report component, without the need to write code to add these functions to each “TfrxReport” and without the need to write additional code for each report component’s “onUser” function handler.

Writing Custom Wizards

You can extend FastReport's functionality with the help of custom wizards. FastReport, for example, contains the standard “Report Wizard” which is called from the “File >|New…” menu item.

There are two types of wizards supported by FastReport. The first type includes the wizard already mentioned, called from the “File>New…” menu item. The second one includes wizards that can be called from the “Wizards” toolbar.

The base class for any wizard is **“TfrxCustomWizard”**, defined in the “frxClass” file.

```
TfrxCustomWizard = class(TComponent)
Public
  Constructor Create(AOwner: TComponent); override;
  class function GetDescription: String; virtual; abstract;
  function Execute: Boolean; virtual; abstract;
  property Designer: TfrxCustomDesigner read FDesigner;
  property Report: TfrxReport read FReport;
end;
```

To write your own wizard inherit from this class and override at least the “GetDescription” and “Execute” methods. The first method returns the wizard name; the second method is called when running the wizard; it must return “True” if the wizard finished successfully and made any changes to the report. While the wizard is running you can access designer and report methods and properties as normal using the “Designer” and “Report” properties.

Wizard registration and deletion is performed using the procedures defined in the “frxDsgnIntf” file:

```
frxWizards.Register(ClassRef: TfrxWizardClass; ButtonBmp: TBitmap;
                    IsToolbarWizard: Boolean = False);
frxWizards.Unregister(ClassRef: TfrxWizardClass);
```

On registration supply the wizard class name, its picture and whether the wizard is to be placed on the “Wizards” toolbar. If the wizard should be placed on the toolbar the ButtonBmp size must be either 16x16 pixels or 32x32 pixels.

Let's look at a primitive wizard that will be accessed through the "File>New..." menu item. It adds a new page to a report.

```
uses frxClass, frxDsgnIntf;

type
  TfrxMyWizard = class(TfrxCustomWizard)
  public
    class function GetDescription: String; override;
    function Execute: Boolean; override;
  end;

class function TfrxMyWizard.GetDescription: String;
begin
  Result := 'My Wizard';
end;

function TfrxMyWizard.Execute: Boolean;
var
```

```
  Page: TfrxReportPage;
begin
  { lock any drawings in designer }
  Designer.Lock;

  { create new page in report }
  Page := TfrxReportPage.Create(Report);
  { create unique name for page }
  Page.CreateUniqueName;
  { set paper sizes and orientation to defaults }
  Page.SetDefaults;

  { update report pages and switch focus to last added page }
  Designer.ReloadPages(Report.PagesCount - 1);
end;

var
  Bmp: TBitmap;

initialization
  Bmp := TBitmap.Create;
  { load picture from resource; of course, it must already be there }
  Bmp.LoadFromResourceName(hInstance, 'frxMyWizard');
  frxWizards.Register(TfrxMyWizard, Bmp);

finalization
  frxWizards.Unregister(TfrxMyWizard);
  Bmp.Free;

end.
```