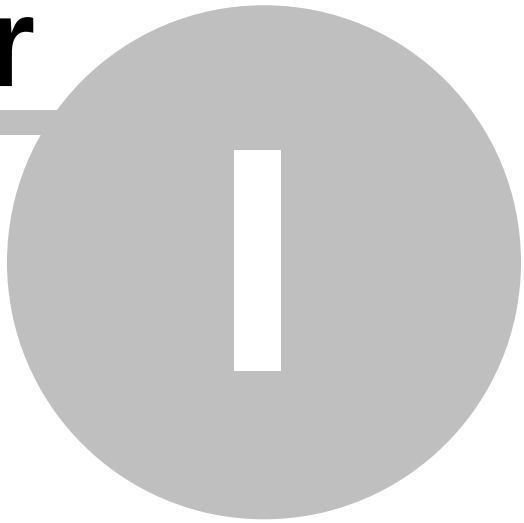
FastReport 4 Programmer's manual

Manual Version 1.2.0

Table of contents

Chapter



Working with the TfrxReport component

1.1 Loading and saving a report

By default, a report is stored together with the project form, i.e. in a DFM file. In most cases nothing else is required and you would not need to take any action to load the report. If you decide to store a report form in a file or in a BLOB-field in a DB (which improves flexibility in that the report can be modified without having to recompile the program) you would have to use “TfrxReport” methods for loading or saving the report:

```
function LoadFromFile(const FileName: String;
                      ExceptionIfNotFound: Boolean = False): Boolean;
```

Loads a report from the file with the specified name. If the second parameter equals “True” and the file is not found then an exception is raised. If the file is loaded successfully it returns “True”.

```
procedure LoadFromStream(Stream: TStream);
```

Loads a report from a stream.

```
procedure SaveToFile(const FileName: String);
```

Saves a report to a file with the specified name.

```
procedure SaveToStream(Stream: TStream);
```

Saves a report to a stream.

Report files are given the default extension “fr3”.

Examples:

Pascal:

```
frxReport1.LoadFromFile('c:\1.fr3');
frxReport1.SaveToFile('c:\2.fr3');
```

C++:

```
frxReport1->LoadFromFile("c:\\1.fr3");
frxReport1->SaveToFile("c:\\2.fr3");
```

1.2 Designing a report

The report designer can be opened using the “TfrxReport.DesignReport” method. The designer must be included in your project (either add the “TfrxDesigner” component to your form or add the “frxDesgn” unit to the “uses” list).

The “DesignReport” method has two default parameters:

```
procedure DesignReport(Modal: Boolean = True; MDIChild: Boolean = False);
```

The Modal parameter determines whether the designer is modal. The MDIChild parameter makes the designer window an MDI child window.

Example:

```
frxReport1.DesignReport;
```

1.3 Running a report

Calling one of the following two “TfrxReport” methods starts (runs, builds) a report:

```
procedure ShowReport(ClearLastReport: Boolean = True);
```

Starts a report and displays the result in the preview window. If the “ClearLastReport” parameter equals “False” then the report will be added to the previously built report, otherwise the previously built report is cleared (by default).

```
function PrepareReport(ClearLastReport: Boolean = True): Boolean;
```

Starts a report without opening the preview window. The parameter has the same function as in the “ShowReport” method. If the report was built successfully it returns “True.”

In most cases it is more convenient to use the first method. It opens the preview window immediately, while the report is being built.

The “ClearLastReport” parameter is useful when reports are to be added one to another, as happens in batch report printing.

Example:

```
frxReport1.ShowReport;
```

1.4 Previewing a report

A report can be displayed in the preview window in two ways: either by calling the “TfrxReport. ShowReport” method (described above) or by calling the “TfrxReport.ShowPreparedReport” method. The second method does not build the report but will display a finished report. This means that the report should either have been built beforehand, using the “PrepareReport” method, or loaded from a previously built report which has been saved in a file (see “Loading and saving a report”).

Example:

Pascal:

```
if frxReport1.PrepareReport then
  frxReport1.ShowPreparedReport;
```

C++:

```
if(frxReport1->PrepareReport(true))
  frxReport1->ShowPreparedReport();
```

Using this code the report is built first before being displayed in the preview window. Building a large report can take a significant time. It is better to use the “ShowReport” method rather than “PrepareReport” and “ShowPreparedReport” as “ShowReport” gives visual feedback of the report building. Settings for the report preview can be made in the “TfrxReport.PreviewOptions” property.

1.5 Printing a report

In most cases a report will be printed from the preview window, but to print a report manually use the “TfrxReport.Print” method, for example:

```
frxReport1.LoadFromFile(...);
frxReport1.PrepareReport;
frxReport1.Print;
```

When “Print” is called the Printer dialogue is opened, in which the printing parameters can be set. The Printer dialogue can be disabled and the printing parameters set using the “TfrxReport.PrintOptions” property.

1.6 Loading and saving a finished report

Loading and saving a finished report can be performed from the preview window. It can also be performed manually using the “TfrxReport.PreviewPages” methods:

```
function LoadFromFile(const FileName: String;
                      ExceptionIfNotFound: Boolean = False): Boolean;
procedure SaveToFile(const FileName: String);
procedure LoadFromStream(Stream: TStream);
procedure SaveToStream(Stream: TStream);
```

The parameters have functions similar to the corresponding TfrxReport methods. Files containing finished reports have an “fp3” extension by default.

Example:

Pascal:

```
frxReport1.PreviewPages.LoadFromFile('c:\1.fp3');
frxReport1.ShowPreparedReport;
```

C++:

```
frxReport1->PreviewPages->LoadFromFile("c:\\1.fp3");
frxReport1->ShowPreparedReport();
```

Note that after the finished report has fully loaded it is previewed by calling the “ShowPreparedReport” method!

1.7 Exporting a report

Reports can be exported from the preview window. They can also be exported manually using the “TfrxReport.Export” method. The export filter to be used should be passed in the method's parameter:

```
frxReport1.Export(frxHTMLExport1);
```

The export filter component must be available and correctly configured (the component must be placed on the project form).

1.8 Creating a custom preview window

FastReport displays reports in the standard preview window. If for any reason this is not satisfactory then a custom preview form can be created. The “TfrxPreview” component from the FastReport component palette was designed for this purpose. To display a report the “TfrxReport.Preview” property should be directed to this “TfrxPreview” component.

There are typically two problems when using the “TfrxPreview” component. It does not always respond to key presses (arrows, PgUp, PgDown etc) and does not respond to the mouse wheel (if present). To make “TfrxPreview” respond to keys pass the focus to it (for example in the OnShow event handler of the form):

```
frxPreview.SetFocus;
```

To make “TfrxPreview” respond to mouse scrolling an “OnMouseWheel” event handler has to be created and the “TfrxPreview.MouseWheelScroll” method called in this handler:

```
procedure TForm1.FormMouseWheel(Sender: TObject; Shift: TShiftState;
                                WheelDelta: Integer;
                                MousePos: TPoint; var Handled: Boolean);
begin
  frxPreview1.MouseWheelScroll(WheelDelta);
end;
```

1.9 Building a composite report (batch printing)

Sometimes a group of reports need to be printed at the same time or displayed in one preview window. FastReport has the means of adding a second report to the end of an already built report. The “TfrxReport.PrepareReport” method has an optional “ClearLastReport” Boolean parameter which defaults to “True”. This parameter, when “True”, clears the output from the preceding built report. The following code shows how to build a batch of two reports:

Pascal:

```
frxReport1.LoadFromFile('1.fr3');
frxReport1.PrepareReport;
frxReport1.LoadFromFile('2.fr3');
frxReport1.PrepareReport(False);
frxReport1.ShowPreparedReport;
```

C++:

```
frxReport1->LoadFromFile("1.fr3");
frxReport1->PrepareReport(true);
frxReport1->LoadFromFile("2.fr3");
frxReport1->PrepareReport(false);
frxReport1->ShowPreparedReport();
```

We load the first report and build it without display in the preview window. Then we load the second report into the same “TfrxReport” object and this time build it with the “ClearLastReport” parameter set to “False”. This allows the second report to be added to the first one built. Finally we display the finished reports in the preview window.

1.9.1 Numbering of pages in a composite report

The “Page”, “Page#”, “TotalPages” and “TotalPages#” system variables can be used to display a page number or total number of pages. In composite reports these variables return the following values:

- *Page* page number in the current report
- *Page#* page number in the batch
- *TotalPages* total number of pages in the current report (the report must be a two-pass one)
- *TotalPages#* total number of pages in the batch

1.9.2 Combining pages into a composite report

As said previously, the “PrintOnPreviousPage” property of the report design page allows pages to be joined together when printing, using up any free space at the end of the previous page to produce a composite report. This property also allows a new report to start printing on any free space at the end of the preceding report’s last page. To do this the “PrintOnPreviousPage” property should be set on the first design page of each succeeding report.

1.10 Interactive reports

Interactive reports can respond to mouse-clicks on any specified report objects in the preview window. For example, clicking on the data line may run a new report with detailed data for the selected line.

Any report can be made to be interactive by creating a “TfrxReport.OnClickObject” event handler, for example:

Pascal:

```
procedure TForm1.frxReport1ClickObject(Page: TfrxPage; View: TfrxView;
                                       Button: TMouseButton;
                                       Shift: TShiftState;
                                       var Modified: Boolean);
begin
```

```
  if View.Name = 'Memo1' then
    ShowMessage('Memo1 contents:' + #13#10 + TfrxMemoView(View).Text);
  if View.Name = 'Memo2' then
  begin
    TfrxMemoView(View).Text := InputBox('Edit', 'Edit Memo2 text:',
                                        TfrxMemoView(View).Text);
    Modified := True;
  end;
end;
```

C++:

```
void __fastcall TForm1::frxReport1ClickObject(TfrxView *Sender,
                                              TMouseButton Button,
                                              TShiftState Shift,
                                              bool &Modified)
{
  TfrxMemoView * Memo;
  if(Memo =  dynamic_cast <TfrxMemoView *> (Sender))
  {
    if(Memo->Name == "Memo1")
      ShowMessage("Memo1 contents:\n\r" + Memo->Text);
    if(Memo->Name == "Memo2")
    {
      Memo->Text = InputBox("Edit", "Edit Memo2 text:", Memo->Text);
       Modified = true;
    }
  }
}
```

In the “OnClickObject” handler, you can do the following:

- modify the contents of an object or a page passed into the handler (the “Modified” output parameter should be set so that the changes are implemented)
- call the “TfrxReport.PrepareReport” method to rebuild a report

In the example, clicking on the “Memo1” object results in the display of a message showing the contents of the object. When clicking on the “Memo2” object a dialogue is displayed where the contents of this object can be changed. Setting the “Modified” flag to “True” makes the change permanent.

In the same way other responses can be made, such as running a new report. PLEASE NOTE : in FastReport v3 and above the TfrxReport component can only display one report in the preview window (unlike FastReport version 2.x). So either the new report must be run in a separate TfrxReport object or the current report must be erased from the current TfrxReport object.

The user can be given a visual clue to clickable objects by changing the mouse cursor when it passes over the objects in the preview window. To do this, select the object in the report designer and set its cursor property to something other than crDefault.

There is one more concern with making objects clickable. Simple reports can test either the object’s name or the object's content to evoke a response. However, this is not always so straight forward in more complicated reports. For example, in a master-detail report when clicking on the “Memo1” object having content '12', where does the data come from? The primary key identifies this line unambiguously. FastReport provides a “TagStr” property for storing any useful string data (in our case the primary key).

Let's illustrate this with a report included in the FastReportDemo.exe example - a 'Simple list'

demo. It is a list of company clients, with data such as “client name”, “address”, “contact person” etc. The data source is the “Customer.db” table from the DBDEMOS demo database. This table has a primary key, the “CustNo” field, which is not displayed in the report output. Our problem is to determine which record is referred to when clicking on any object in the finished report. The value of the primary key is required, which can be entered as an expression into the “TagStr” property of all the objects in the MasterData band:

```
[Customers."CustNo"]
```

When a report is being built the “TagStr” property’s contents are evaluated in the same way as for the contents of text objects, that is values are substituted for named variables and expressions (enclosed in square brackets) are evaluated. That is why the “TagStr” property of the objects lying on the MasterData band contains values such '1005' or '2112', etc. after report building. A simple conversion from a string to an integer gives us the value of the primary key from which the required record can be found.

If the primary key is composite (i.e. it contains more than one field) the “TagStr” property’s contents may be the following:

```
[Table1."Field1"];[Table1."Field2"]
```

After building the report the “TagStr” property may contain a string value of '1000;1', from which the individual field values can be fairly simply extracted.

1.11 Accessing report objects from code

FastReport’s objects (report page, band, memo object, etc.) are not directly accessible from your code. This means that you cannot address an object directly by its name, as for example when addressing a button on your form. To address an object it should be first be found using the “TfrxReport.FindObject” method:

Pascal:

```
var
  Memo1: TfrxMemoView;

Memo1 := frxReport1.FindObject('Memo1') as TfrxMemoView;
```

C++:

```
TfrxMemoView * Memo = dynamic_cast <TfrxMemoView *>
                          (frxReport1->FindObject("Memo1"));
```

Once found the object’s properties and methods can be accessed.

You can address the report’s pages using the “TfrxReport.Pages” property:

Pascal:

```
var
  Page1: TfrxReportPage;

Page1 := frxReport1.Pages[1] as TfrxReportPage;
```

C++:

```
TfrxReportPage * Page1 = dynamic_cast <TfrxReportPage *>
                           (frxReport1->Pages[1]);
```

1.12 Creating a report form from code

As a rule you will create most reports using the designer. However, sometimes it is necessary to create a report manually in code, for example when the report's form is unknown.

To create a report manually requires the following steps in order:
- clear the report component
- add data sources
- add the "Data" page
- add the report page
- add bands on the page
- set band properties and then connect them to the data
- add objects on each band
- set object properties and then connect them to the data

Let's look at the creation of a simple report of "list" type. Assume we have the following components: frxReport1: TfrxReport and frxDBDataSet1: TfrxDBDataSet (the latter connected to data from the DBDEMOS "Customer.db" table). Our report will contain one page with "ReportTitle" and "MasterData" bands. On the "ReportTitle" band there will be an object containing "Hello FastReport!" and the "MasterData" band will contain an object linked to the "CustNo" field.

Pascal:

```
var
  DataPage: TfrxDataPage;
  Page: TfrxReportPage;
  Band: TfrxBand;
  DataBand: TfrxMasterData;
  Memo: TfrxMemoView;

{ clear report }
frxReport1.Clear;

{ add dataset to list of datasets accessible in report }
frxReport1.DataSets.Add(frxDBDataSet1);

{ add "Data" page }
DataPage := TfrxDataPage.Create(frxReport1);

{ add page }
Page := TfrxReportPage.Create(frxReport1);
{ create a unique name }
Page.CreateUniqueName;
{ set sizes of fields, paper and orientation to defaults }
Page.SetDefaults;
{ change paper orientation }
Page.Orientation := poLandscape;

{ add report title band}
Band := TfrxReportTitle.Create(Page);
Band.CreateUniqueName;
{ only "Top" coordinate and height of band need setting
```

```
    both in pixels }
Band.Top := 0;
Band.Height := 20;

{ add object to report title band }
Memo := TfrxMemoView.Create(Band);
Memo.CreateUniqueName;
Memo.Text := 'Hello FastReport!';
Memo.Height := 20;
{ this object will be stretched to band's width }
Memo.Align := baWidth;

{ add masterdata band }
DataBand := TfrxMasterData.Create(Page);
DataBand.CreateUniqueName;
DataBand.DataSet := frxDBDataSet1;
{ "Top" should be greater than previously added band's top + height }
DataBand.Top := 100;
DataBand.Height := 20;

{ add object on masterdata }
Memo := TfrxMemoView.Create(DataBand);
Memo.CreateUniqueName;
{ connect to data }
Memo.DataSet := frxDBDataSet1;
Memo.DataField := 'CustNo';
Memo.SetBounds(0, 0, 100, 20);
{ align text to object's right margin }
Memo.HAlign := haRight;

{ show report }
frxReport1.ShowReport;
```

C++:

```
TfrxDataPage * DataPage;
TfrxReportPage * Page;
TfrxBand * Band;
TfrxMasterData * DataBand;
TfrxMemoView * Memo;

// clear report
frxReport1->Clear();

// add dataset to list of datasets accessible in report
frxReport1->DataSets->Add(frxDBDataset1);

// add "Data" page
DataPage = new TfrxDataPage(frxReport1);

// add page
Page = new TfrxReportPage(frxReport1);
// create unique name
Page->CreateUniqueName();
// set sizes of fields, paper and orientation to defaults
Page->SetDefaults();
// change paper orientation
Page->Orientation = poLandscape;

// add report title band
Band = new TfrxReportTitle(Page);
Band->CreateUniqueName();
// only "Top" coordinate and height of band need setting
```

```
//    both in pixels
Band->Top = 0;
Band->Height = 20;

// add object to report title band
Memo = new TfrxMemoView(Band);
Memo->CreateUniqueName();
Memo->Text = "Hello FastReport!";
Memo->Height = 20;
// this object will be stretched to band’s width
Memo->Align = baWidth;

// add masterdata band
DataBand = new TfrxMasterData(Page);
DataBand->CreateUniqueName();
DataBand->DataSet = frxDBDataset1;
// “Top” should be greater than previously added band’s top + height
DataBand->Top = 100;
DataBand->Height = 20;

// add object on masterdata
Memo = new TfrxMemoView(DataBand);
Memo->CreateUniqueName();
// connect to data
Memo->DataSet = frxDBDataset1;
Memo->DataField = "CustNo";
Memo->SetBounds(0, 0, 100, 20);
// align text to object’s right margin
Memo->HAlign = haRight;

// show report
frxReport1->ShowReport(true);
```

Let's explain some details.

All the data sources that are to be used in the report must be added to the list of data sources, otherwise the report will not work. In our case use “frxReport1.DataSets.Add(frxDBDataSet1)”

The “Data” page is required when inserting internal datasets, such as “TfrxADOTable”, into a report. Such datasets can only be placed on the “Data” page.

The call to Page.SetDefaults is not essential since in this case the page will be A4 format with margins of 0 mm. SetDefaults sets margins to 10mm and page size and alignment to the default printer's values.

When adding bands to a page make sure that they do not overlap each other. To ensure this set the “Top” and “Height” properties. There is no point in changing the “Left” and “Width” properties since a band always has the same width as the page on which it is located : if the bands are vertical this is not so – instead set the “Left” and “Width” properties and don't bother with the “Top” and “Height” properties. Note that the ordering of the bands on the page is critical. Always locate bands in the same order as you would do in the designer.

Object coordinates and sizes are set in pixels. Since the “Left”, “Top”, “Width” and “Height” properties of all objects are of “Extended” type you can set non-integer values. The following constants are defined for converting pixels into centimeters or inches:

```
fr01cm = 3.77953;
fr1cm  = 37.7953;
fr01in = 9.6;
fr1in  = 96;
```

So for example, a band's height can be set to 5 mm as follows:

```
Band.Height := fr01cm * 5;
Band.Height := fr1cm * 0.5;
```

1.13 Creating a dialogue form from code

As we know a report can contain dialogue forms. The following example shows how to create a dialogue form with an "OK" button:

Pascal:

```
{ to work with dialogue objects the following unit must be used }
uses frxDCtrl;

var
  Page: TfrxDialogPage;
  Button: TfrxButtonControl;

{ add page }
Page := TfrxDialogPage.Create(frxReport1);
{ create unique name }
Page.CreateUniqueName;
{ set sizes }
Page.Width := 200;
Page.Height := 200;
{ set position }
Page.Position := poScreenCenter;

{ add button }
Button := TfrxButtonControl.Create(Page);
Button.CreateUniqueName;
Button.Caption := 'OK';
Button.ModalResult := mrOk;
Button.SetBounds(60, 140, 75, 25);

{ show report }
frxReport1.ShowReport;
```

C++:

```
// to work with dialogue objects the following unit must be used
#include "frxDCtrl.hpp"

TfrxDialogPage * Page;
TfrxButtonControl * Button;

// add page
Page = new TfrxDialogPage(frxReport1);
// create unique name
Page->CreateUniqueName();
// set sizes
Page->Width = 200;
Page->Height = 200;
// set position
Page->Position = poScreenCenter;

// add button
```

```
Button = new TfrxButtonControl(Page);
Button->CreateUniqueName();
Button->Caption = "OK";
Button->ModalResult = mrOk;
Button->SetBounds(60, 140, 75, 25);

// show report
frxReport1->ShowReport(true);
```

1.14 Modifying a report page's properties

Sometimes it is necessary to change report page settings (for example paper alignment or size) in code. The "TfrxReportPage" class contains the following properties which define the size of the page:

```
property Orientation: TPrinterOrientation default poPortrait;
property PaperWidth: Extended;
property PaperHeight: Extended;
property PaperSize: Integer;
```

The "PaperSize" property sets the paper format. This is one of the standard values defined in Windows.pas (for example DMPAPER_A4). If a value is assigned to this property then FastReport sets the "PaperWidth" and "PaperHeight" properties automatically (paper size in millimeters). Setting "PaperSize" to DMPAPER_USER (or to 256) means that a custom paper size is set. In this case the "PaperWidth" and "PaperHeight" properties will need to be be set in code.

The following example shows how to change the properties of the first page, assuming we already have a report:

Pascal:

```
var
  Page: TfrxReportPage;

{ first report page has index [1] : index [0] is the Data page }
Page := TfrxReportPage(frxReport1.Pages[1]);
{ change size }
Page.PaperSize := DMPAPER_A2;
{ change  paper orientation }
Page.Orientation := poLandscape;
```

C++:

```
TfrxReportPage * Page;

// first report page has index [1] : index [0] is the Data page
Page = (TfrxReportPage *)frxReport1.Pages[1];
// change size
Page->PaperSize = DMPAPER_A2;
// change paper orientation
Page->Orientation = poLandscape;
```

1.15 Constructing a report with the help of code

The FastReport engine usually is responsible for the building of reports. It displays the report bands in the specified order as many times as the data source to which it is connected requires until the report is finished. Sometimes it is necessary to create a report in a non-standard form that the FastReport engine is unable to produce. In this case the report can be built manually, using the “TfrxReport.OnManualBuild” event. When a handler is defined for this event the FastReport engine hands some management functions over to it. The allocation of responsibilities for building the report is changed to the following:

FastReport engine:
- report preparation (script, data sources initialization, band tree formation)
- all calculations (aggregate functions, event handlers)
- creation of new pages and columns (automatic display of a page and column headers and footers and of report title and summary)
- other routine work

Handler:
- ordering of the bands

The purpose of the “OnManualBuild” handler is to issue commands for displaying particular bands to the FastReport engine. The engine itself does the rest : creating new pages as soon as there is no free space left in the current one, execution of scripts, etc.

The engine is represented by the “TfrxCustomEngine” class. A link to the instance of this class is located in the “TfrxReport.Engine” property.

```
procedure NewColumn;
```
Creates a new column. If a column is the last one on the page then it creates a new page.

```
procedure NewPage;
```
Creates a new page.

```
procedure ShowBand(Band: TfrxBand); overload;
```
Displays a band.

```
procedure ShowBand(Band: TfrxBandClass); overload;
```
Displays a band of the given type.

```
function FreeSpace: Extended;
```
Returns the amount of free space on the page (in pixels). This value is decremented after the next band has been displayed.

```
property CurColumn: Integer;
```
Returns or sets the current column number.

```
property CurX: Extended;
```
Returns or sets the current X position.

```
property CurY: Extended;
```
Returns or sets the current Y position. This value is incremented after the next band has been displayed.

```
property DoublePass: Boolean;
```
Defines whether a report is a two-pass one.

`property FinalPass: Boolean;`
Returns whether the current pass is the last one.

`property FooterHeight: Extended;`
Returns the page footer height.

`property HeaderHeight: Extended;`
Returns the page header height.

`property PageHeight: Extended;`
Returns the height of the page's printable region.

`property PageWidth: Extended;`
Returns the width of the page's printable region.

`property TotalPages: Integer;`
Returns the number of pages in a finished report (only in the second pass of a two-pass report).

Let's show an example of a simple handler. There are two "MasterData" bands in a report, which are not connected to data. The handler displays these bands in an interlaced order, six times for each one. After six bands a small gap is introduced.

Pascal:

```
var
  i: Integer;
  Band1, Band2: TfrxMasterData;

{ find specified bands }
Band1 := frxReport1.FindObject('MasterData1') as TfrxMasterData;
Band2 := frxReport1.FindObject('MasterData2') as TfrxMasterData;

for i := 1 to 6 do
begin
  { display bands one after another }
  frxReport1.Engine.ShowBand(Band1);
  frxReport1.Engine.ShowBand(Band2);
  { introduce a small gap }
  if i = 3 then
    frxReport1.Engine.CurY := frxReport1.Engine.CurY + 10;
end;
```

C++:

```
int i;
TfrxMasterData * Band1;
TfrxMasterData * Band2;

// find specified bands
Band1 := dynamic_cast <TfrxMasterData *> (frxReport1->FindObject
("MasterData1"));
Band2 := dynamic_cast <TfrxMasterData *> (frxReport1->FindObject
("MasterData2"));

for(i = 1; i <= 6; i++)
{
  // display bands one after another
  frxReport1->Engine->ShowBand(Band1);
```

```
  frxReport1->Engine->ShowBand(Band2);
  // introduce a small gap
  if(i == 3)
    frxReport1->Engine->CurY += 10;
}
```

The next example shows two groups of bands side by side.

Pascal:

```
var
  i, j: Integer;
  Band1, Band2: TfrxMasterData;
  SaveY: Extended;

Band1 := frxReport1.FindObject('MasterData1') as TfrxMasterData;
Band2 := frxReport1.FindObject('MasterData2') as TfrxMasterData;

SaveY := frxReport1.Engine.CurY;
for j := 1 to 2 do
begin
  for i := 1 to 6 do
  begin
    frxReport1.Engine.ShowBand(Band1);
    frxReport1.Engine.ShowBand(Band2);
    if i = 3 then
      frxReport1.Engine.CurY := frxReport1.Engine.CurY + 10;
  end;
  frxReport1.Engine.CurY := SaveY;
  frxReport1.Engine.CurX := frxReport1.Engine.CurX + 200;
end;
```

C++:

```
int i, j;
TfrxMasterData * Band1;
TfrxMasterData * Band2;
Extended SaveY;

Band1 = dynamic_cast <TfrxMasterData *> (frxReport1->FindObject
("MasterData1"));
Band2 = dynamic_cast <TfrxMasterData *> (frxReport1->FindObject
("MasterData2"));

SaveY = frxReport1->Engine->CurY;
for(j = 1; j <= 2; j++)
{
  for(i = 1; i <= 6; i++)
  {
    frxReport1->Engine->ShowBand(Band1);
    frxReport1->Engine->ShowBand(Band2);
    if(i == 3)
      frxReport1->Engine->CurY += 10;
  }
  frxReport1->Engine->CurY = SaveY;
  frxReport1->Engine->CurX += 200;
}
```

1.16 Printing an array

The code for this example is located in the “FastReport Demos\PrintArray” ("FastReport Demos\BCB Demos\PrintArray") folder. Let's explain some details of this code.

To print an array we use a report having one “MasterData” band which will be displayed as many times as there are elements in the array. To do this place a “TfrxUserDataSet” component on the form and then set these properties (this can be done in code, as shown in the example):

```
RangeEnd := reCount
RangeEndCount := a number of elements in an array
```

After that connect the data band to the “TfrxUserDataSet” component. To represent an array element place a text object containing the expression “[<element>]” inside the “MasterData” band. The “element” variable is filled using the “TfrxReport.OnGetValue” event.

1.17 Printing a TStringList

The code for this example is located in the “FastReport Demos\PrintStringList” (“FastReport Demos\BCB Demos\PrintStringList”) folder. The method is the same as for the example of printing an array.

1.18 Printing a file

The code for this example is located in the “FastReport Demos\PrintFile” (“FastReport Demos\BCB Demos\PrintFile”) older. Let's explain some details of this code.

For printing, the report should contain a “MasterData” band that will be printed just once (to do this connect the band to a data source that contains just one record; select the source named "Single row" from the list). Stretching (“Stretch”) and splitting (“Allow Split”) are enabled for the band, which means that the band is stretched to make room for all the objects located on it and if the page has insufficient room for the band then the band is split over two or more pages.

File contents are displayed using a “Text” object containing the expression “[<file>]” variable. This variable, as in previous examples, is filled using the “TfrxReport.OnGetValue” event. Stretching is also enabled for the object (“Stretch” from the contextual menu or “StretchMode” property = smActualHeight).

1.19 Printing a TStringGrid

The code for this example is located in the “FastReport Demos\PrintStringGrid” (“FastReport Demos\BCB Demos\PrintStringGrid”) folder. Let's explain some details of this code.

The “TStringGrid” component represents a table having several rows and columns. This means that a report expands not only in height but also in width. Let's use the “Cross-tab” object to print this component (available when a “TfrxCrossObject” component is added to the project). The “Cross-tab” object is responsible only for printing table data with an unknown number of

rows and columns. The object has two versions: “TfrxCrossView” to print custom data in code, and “TfrxDBCrossView” to print special kind of data from a DB table.

Let's use a “TfrxCrossView”. The object must be initialized, by first opening the report designer and then the object's editor by double-clicking on it. The number of rows and columns must be set, and also the number of values in the table cells. We will use '1' for all of these values and will disable the row and column titles and the totals for lines and columns.

The object must be filled with values from the StringGrid in the “TfrxReport.OnBeforePrint” event. Values are added using the “TfrxCrossView.AddValue” method, whose parameters are: composite index for a line, a column and the cell’s value (which is composite as well, as an object can contain more than one value in a cell).

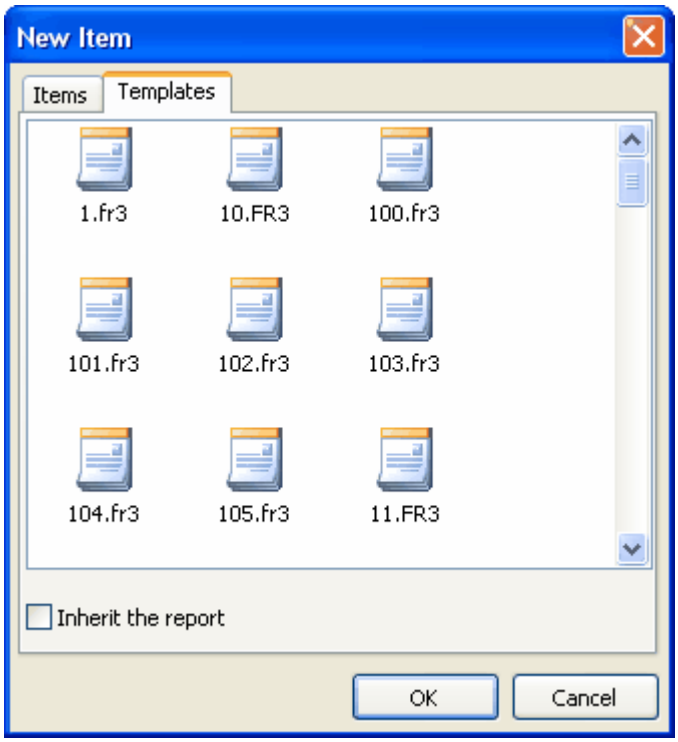
1.20 Printing a TTable or TQuery

The code for this example is located in the “FastReport’s Demos\PrintTable” (“FastReport Demos\BCB Demos\PrintTable”) folder. The principles are the same as for the example of printing a TStringGrid. In this case, the row index is its sequence number, the column index is the name of a table field and the cell value is the table's field value. It is important to note that the functions for cell elements must be disabled in the “Cross-tab” object editor (since cells can contain data of various types, which leads to an error during table creation) and the table title sorting must also be disabled (otherwise the columns will be sorted alphabetically).

1.21 Report inheritance

Report inheritance is described in the User's manual. We will mention some key points here.

If your reports are stored in files then FastReport needs to be told which folder to search for base reports. This folder's content is displayed in the “File>New...” and “Report>Options...” dialogs:



The “TfrxDesigner.TemplateDir” property is used for this purpose. By default it is empty and FastReport searches for base reports in the same folder as the project's executable file (.exe). An absolute or a relative path can be set in this property.

If your reports are stored in a database then code must be written to get a list of available base reports from the DB and to load the base report from the DB. Use “TfrxReport. OnLoadTemplate” event to load a base report:

```
property OnLoadTemplate: TfrxLoadTemplateEvent read FOnLoadTemplate
                                                write FOnLoadTemplate;

TfrxLoadTemplateEvent = procedure(Report: TfrxReport;
                                  const TemplateName: String) of object;
```

This event handler must load a base report with the specified TemplateName into the Report object. Here's an example of a handler:

```
procedure TForm1.LoadTemplate(Report: TfrxReport;
                              const TemplateName: String);
var
  BlobStream: TStream;
begin
  ADOTable1.First;
  while not ADOTable1.Eof do
  begin
    if AnsiCompareText(ADOTable1.FieldByName('ReportName').AsString,
                       TemplateName) = 0 then
    begin
      BlobStream := TMemoryStream.Create;
      TBlobField(ADOTable1.FieldByName('ReportBlob'))
                 .SaveToStream(BlobStream);
      BlobStream.Position := 0;
      Report.LoadFromStream(BlobStream);
```

```
        BlobStream.Free;
        break;
      end;
      ADOTable1.Next;
    end;
  end;
```

To get a list of available templates use the “TfrxDesigner.OnGetTemplateList” event:

```
  property OnGetTemplateList: TfrxGetTemplateListEvent
                                    read FOnGetTemplateList
                                    write FOnGetTemplateList;

  TfrxGetTemplateListEvent = procedure(List: TStrings) of object;
```

This event handler returns a list of available templates in the List parameter. Here's an example of a handler:

```
  procedure TForm1.GetTemplates(List: TList);
  begin
    List.Clear;
    ADOTable1.First;
    while not ADOTable1.Eof do
    begin
      List.Add(ADOTable1.FieldByName('ReportName').AsString);
      ADOTable1.Next;
    end;
  end;
```

Fast Report can inherit from previously created reports. For this use the function:

```
   TfrxReport.InheritFromTemplate(const templName: String;
                                  InheritMode: TfrxInheritMode
                                               = imDefault): Boolean
```

This function makes the current loaded report inherit from the specified template. The first parameter is the name and path of the parent template, the second sets the inherit mode, which is one of :

imDefault (by default)	- open dialogue offering renaming/deletion of duplicates
imDelete	- delete all duplicate objects
imRename	- rename all duplicate objects.

Please Note!
The search for the parent template is done in reference to the current template. FastReport uses relative paths so there is normally no need to worry about moving applications; the only exception is when the current report and the parent template are placed on different folders or a net path is used.

1.22 Multi-threading

FastReport can operate independently in more than one thread, but there are some considerations:

- “TfrxDBDataSet” cannot be created in different threads because the “global list” is used for

searching and the dataset will always be taken from the first created “TfrxDBDataSet” (use of the global list can be switched off - it is active by default)

- if there are changes in object properties (for example Memo1.Left := Memo1.Left + 10 in script) during report execution then it must be remembered that during the next operation, if “TfrxReport.EngineOptions.DestroyForms = False” the report template will already have been changed and will need to be reloaded, or use “TfrxReport.EngineOptions. DestroyForms := True”. During renewal you can't use interactive reports from the thread because the script's objects are deleted after renewal, that is why in some cases it is better to use “TfrxReport.EngineOptions.DestroyForms := False” and renew the template on your own during the next build cycle

If required the global list which is searched for copies of “TfrxDBDataSet” can be switched off:

```
{ DestroyForms can be switched off, if every time a report is renewed
  from a file or from the current report }
FReport.EngineOptions.DestroyForms := False;
FReport.EngineOptions.SilentMode := True;

{ This property switches off the search of the global list }
FReport.EngineOptions.UseGlobalDataSetList := False;

{EnabledDataSets plays the role of a local list, it should be installed
 before the template is loaded }
FReport.EnabledDataSets.Add(FfrxDataSet);
FReport.LoadFromFile(ReportName);
FReport.PrepareReport;
```

1.23 Report caching

A report and its data can be cached both in memory (to increase speed) and in a file on disk (to reduce RAM usage). There are several types of caching in Fast Report:

- “TfrxReport.EngineOptions.UseFileCache” - when True the whole text and objects of a built report are saved in a temporary file on disk; “TfrxReport.EngineOptions.MaxMemoSize” sets how many MB are reserved for the report in RAM

- “TfrxReport.PreviewOptions.PagesInCache” - the number of pages which can be kept in cached memory, which greatly increases preview speed, but uses a lot of memory (especially if there are pictures in the report)

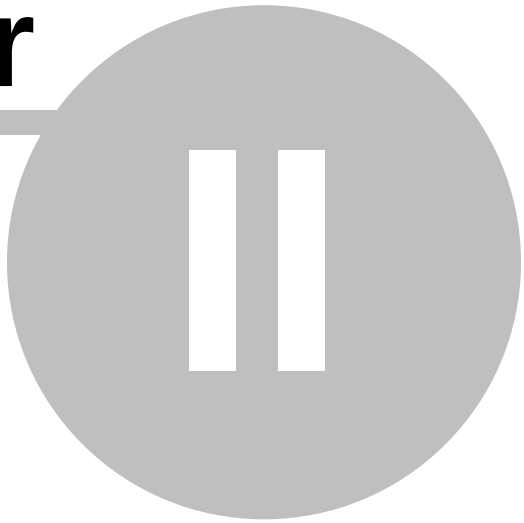
- “TfrxReport.PreviewOptions.PictureCacheInFile” - when True all pictures in a built report are saved in a temporary file on disk, which greatly reduces memory use in reports that have a large number of pictures; but it reduces the speed

1.24 MDI architecture

In Fast Report there is opportunity for creating MDI applications, both for preview and with a designer. The code for an example is located in “FastReport Demos\MDI Designer catalogue”.

It is worth mentioning that it is advisable to create a “TfrxReport” for each preview window or designer, otherwise all windows will refer to a single report.

Chapter



Working with a list of variables

The notion of variables was explained in detail in the corresponding chapter. Let's briefly call to mind the main points.

A user can specify one or more variables in a report. A value or an expression, which will be automatically calculated when referring to a variable, can be assigned to every variable. Variables can be visually inserted into a report from the “Data tree” pane. It is convenient to use variables as aliases for compound expressions, which are often used in reports.

The “frxVariables” unit must be used in the project when working with variables. Variables are represented by the “TfrxVariable” class.

```
TfrxVariable = class(TCollectionItem)
published
  property Name: String;
  ( the name of the variable }

  property Value: Variant;
  { the value of the variable }
  end;
```

The list of variables is represented by the “TfrxVariables” class. It contains all the methods necessary for working with the list.

```
TfrxVariables = class(TCollection)
public
  function Add: TfrxVariable;
  { adds a variable to the end of the list }

  function Insert(Index: Integer): TfrxVariable;
  { adds a variable at the given position in the list }

  function IndexOf(const Name: String): Integer;
  { returns the index of the variable with the given name }

  procedure AddVariable(const ACategory, AName: String;
                        const AValue: Variant);
  { adds a variable to the specified category }

  procedure DeleteCategory(const Name: String);
  { deletes a category and all its variables }

  procedure DeleteVariable(const Name: String);
  { deletes a variable }

  procedure GetCategoriesList(List: TStrings; ClearList: Boolean = True);
  { returns the list of categories }

  procedure GetVariablesList(const Category: String; List: TStrings);
  { returns the list of variables in the specified category }

  property Items[Index: Integer]: TfrxVariable readonly;
  { the list of variables }

  property Variables[Index: String]: Variant; default;
  { value of specified variable }

  end;
```

When the list of variables is long it can be convenient to group the variables by categories. For example, having the following list of variables:

Customer name
Account number
in total
total vat

they can be represented it in the following way:

Properties
  Customer name
  Account number
Totals
  in total
  total vat

There are some limitations:

- at least one category must be created
- categories form the first level of the data tree, variables form the second
- categories cannot be nested
- variable names must be unique within the whole list, not just within a category

2.1 Creating a list of variables

A link to the report variables is stored in the “TfrxReport.Variables” property. To create a list manually, the following steps must be taken:

- clear the list
- create a category
- create variables
- repeat the second and third steps to create variables in another category

2.2 Clearing a list of variables

A list of variables is cleared using the “TfrxVariables.Clear” method:

Pascal:

```
frxReport1.Variables.Clear;
```

C++:

```
frxReport1->Variables->Clear();
```

2.3 Adding a category

At least one category must be created. Categories and variables are both stored in the one list. A category differs from a variable by having a “space” character as the first symbol of the name. All variables located in the list after a category are considered as belonging to that category.

A category can be added to the list in either of two ways:

Pascal:

```
frxReport1.Variables[' ' + 'My Category 1'] := Null;
```

C++:

```
frxReport1->Variables->Variables[" My Category 1"] = NULL;
```

or

Pascal:

```
var
  Category: TfrxVariable;

Category := frxReport1.Variables.Add;
Category.Name := ' ' + 'My category 1';
```

C++:

```
TfrxVariable * Category;

Category = frxReport1->Variables->Add();
Category->Name = " My category 1";
```

2.4 Adding a variable

Variables can be added only after a category has already been added. All the variables located in the list after a category are considered belonging to that category. Variable names must be unique within the whole list and not just within a category

There are several ways to add a variable to the list:

Pascal:

```
frxReport1.Variables['My Variable 1'] := 10;
```

C++:

```
frxReport1->Variables->Variables["My Variable 1"] = 10;
```

this way adds a variable (if it doesn't already exist) or changes the value of an existing variable.

Pascal:

```
var
  Variable: TfrxVariable;
```

```
Variable := frxReport1.Variables.Add;
Variable.Name := 'My Variable 1';
Variable.Value := 10;
```

C++:

```
TfrxVariable * Variable;

Variable = frxReport1->Variables->Add();
Variable->Name = "My Variable 1";
Variable->Value = 10;
```

Both of the ways add a variable to the end of the list, so it is added to the last category. If a variable is to be added at a specific position in the list use the “Insert” method:

Pascal:

```
var
  Variable: TfrxVariable;

Variable := frxReport1.Variables.Insert(1);
Variable.Name := 'My Variable 1';
Variable.Value := 10;
```

C++:

```
TfrxVariable * Variable;

Variable = frxReport1->Variables->Insert(1);
Variable->Name = "My Variable 1";
Variable->Value = 10;
```

If a variable is to be added to a specific category use the “AddVariable” method:

Pascal:

```
frxReport1.Variables.AddVariable('My Category 1', 'My Variable 2', 10);
```

C++:

```
frxReport1->Variables->AddVariable("My Category 1", "My Variable 2", 10);
```

2.5 Deleting a variable

Pascal:

```
frxReport1.Variables.DeleteVariable('My Variable 2');
```

C++:

```
frxReport1->Variables->DeleteVariable("My Variable 2");
```

2.6 Deleting a category

To delete a category with all its variables use the following code:

Pascal:

```
frxReport1.Variables.DeleteCategory('My Category 1');
```

C++:

```
frxReport1->Variables->DeleteCategory("My Category 1");
```

2.7 Modifying a variable's value

There are two ways to modify the value of a variable:

Pascal:

```
frxReport1.Variables['My Variable 2'] := 10;
```

C++:

```
frxReport1->Variables->Variables["My Variable 2"] = 10;
```

or

Pascal:

```
var
  Index: Integer;
  Variable: TfrxVariable;

{ search for the variable }
Index := frxReport1.Variables.IndexOf('My Variable 2');
{ if it is found, change a value }
if Index <> -1 then
begin
  Variable := frxReport1.Variables.Items[Index];
  Variable.Value := 10;
end;
```

C++:

```
int Index;
TfrxVariable * Variable;

// search for the variable
Index = frxReport1->Variables->IndexOf("My Variable 2");
// if it is found, change a value
if(Index != -1)
{
  Variable = frxReport1->Variables->Items[Index];
  Variable->Value = 10;
}
```

It should be noted, that when accessing a report variable its value is calculated if it is of string type. That means the variable whose value is 'Table1."Field1"' will return a value of a DB field and not the 'Table1."Field1"' string. You should be careful when assigning a string-type value to

a report variable. For example, this code will raise the exception “unknown variable 'test'” when running a report:

```
frxReport1.Variables['My Variable'] := 'test';
```

because FastReport is trying to calculate a value for variable 'test'. The right way to pass a string value is:

```
frxReport1.Variables['My Variable'] := '''' + 'test' + '''';
```

In this case the variable value, string 'test', will be shown without errors. But keep in mind that:
- a string should not contain single quotes : all single quotes must be doubled;
- a string should not contain #13 or #10 symbols.

In some cases it is easier to pass variables using a script.

2.8 Script variables

Instead of report variables, script variables are in the TfrxReport.Script. You can define them using FastScript methods. Let's look at some differences between report and script variables:

	Report variables	Script variables
Placement	in the report variables list, TfrxReport.Variables	in the report script, TfrxReport.Script.Variables
Variable name	may contain any symbol	may contain any symbol; but if used inside the report script its name must conform to Pascal identificator requirements
Variable value	may be of any type; variables of string type are calculated each time they are accessed, and are, in themselves, expressions	may be of any type; no calculation is performed; behavior is like a standard language variable.
Accessibility	programmer can see the list of report variables in the “Data tree” pane	variable is not visible - programmer must know it exists

Working with script variables is easy. Just assign a value to the variable like this:

Pascal:

```
frxReport1.Script.Variables['My Variable'] := 'test';
```

C++:

```
frxReport1->Script->Variables->Variables["My Variable"] = "test";
```

Here FastReport creates the variable if it does not exist, or assigns the value to an existing variable. There is no need to use extra quotes when assigning strings to variables.

2.9 Passing a variable value in TfrxReport.OnGetValue

The last way to pass a value to a report is to use the “TfrxReport.OnGetValue” event handler. This is convenient when a dynamic value is to be passed to a report (a value that may change from record to record). The two previous ways are suitable for passing static values.

Let's look at an example of using this event handler. Create a report and place a "Text" object in it. Type the following text into this object:

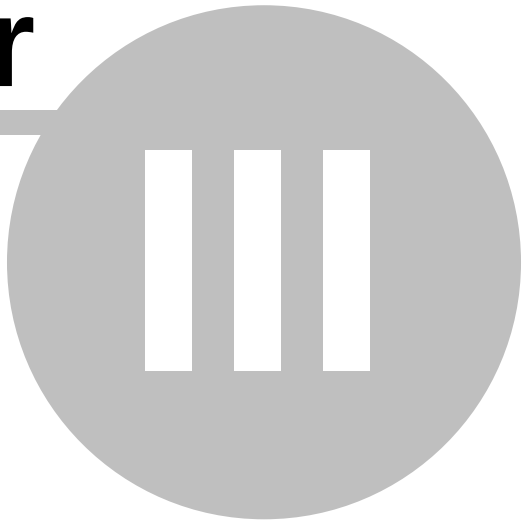
```
[My Variable]
```

Now create the “TfrxReport.OnGetValue” event handler:

```
procedure TForm1.frxReport1GetValue(const VarName: String;
                                    var Value: Variant);
begin
  if CompareText(VarName, 'My Variable') = 0 then
    Value := 'test'
end;
```

Run the report and see that the variable is displayed correctly. The “TfrxReport.OnGetValue” event handler is called each time that FastReport finds an unknown variable. The event handler should return a value for that variable.

Chapter



Working with styles

First of all, let's remember what a “style”, a “set of styles” and a “library of styles” are.

A style is an element that has a name and properties, and that determines some design attributes such as color, font and frame. The style determines the way a report object is displayed. Objects such as “TfrxMemoView” have the Style property, which holds a style name. When a value is given to this property the style design attributes are applied to the object.

A set of styles consists of several styles, which are used in a report. The “TfrxReport” component has the “Styles” property, which points to an object of the “TfrxStyles” type. The set of styles also has a name. The set of styles determines the design appearance of a whole report.

A styles library includes several sets of styles. A specific style set can conveniently be selected for a report from a styles library.

“TfrxStyleItem” represents a style.

```
TfrxStyleItem = class(TCollectionItem)
public
  property Name: String;
  { style name }

  property Color: TColor;
  { background color }

  property Font: TFont;
  { font }

  property Frame: TfrxFrame;
  { frame }
end;
```

A set of styles is represented by the “TfrxStyles” class. It has methods for performing set operations such as reading, saving, adding and deleting, as well as searching for a style. A set of styles file has an “fs3” extension by default.

```
TfrxStyles = class(TCollection)
public
  constructor Create(AReport: TfrxReport);
  { creates the styles set;
    “nil” can be specified instead of “AReport,”
    however a user could not then use the “Apply” method }

  function Add: TfrxStyleItem;
  { adds a new style }

  function Find(const Name: String): TfrxStyleItem;
  { returns the style with the given name }

  procedure Apply;
  { applies a set to a report }

  procedure GetList(List: TStrings);
  { returns the list of style names }
```

```
  procedure LoadFromFile(const FileName: String);
  procedure LoadFromStream(Stream: TStream);
  { reads a set }

  procedure SaveToFile(const FileName: String);
  procedure SaveToStream(Stream: TStream);
  { saves a set }

  property Items[Index: Integer]: TfrxStyleItem; default;
  { the list of styles }

  property Name: String;
  { a set's name }

  end;
```

In conclusion, the "TfrxStyleSheet" class represents a styles library. It has methods for library reading/saving, as well as adding, deleting and searching for style sets.

```
TfrxStyleSheet = class(TObject)
public
  constructor Create;
  { constructs a library }

  procedure Clear;
  { clears a library }

  procedure Delete(Index: Integer);
  { deletes a set with specified index number }

  procedure GetList(List: TStrings);
  { returns the list of names of styles sets }

  procedure LoadFromFile(const FileName: String);
  procedure LoadFromStream(Stream: TStream);
  { loads a library }

  procedure SaveToFile(const FileName: String);
  procedure SaveToStream(Stream: TStream);
  { saves a library }

  function Add: TfrxStyles;
  { adds a new set of styles to the library }

  function Count: Integer;
  { returns the number of styles sets in the library }

  function Find(const Name: String): TfrxStyles;
  { returns a set with the specified name }

  function IndexOf(const Name: String): Integer;
  { returns a set number with the specified name }

  property Items[Index: Integer]: TfrxStyles; default;
  { the list of styles sets }

  end;
```

3.1 Creation of style sets

The following code shows how to create a styles set, with the addition of two styles to the set. After these operations are completed the styles are applied to the report.

Pascal:

```
var
  Style: TfrxStyleItem;
  Styles: TfrxStyles;

Styles := TfrxStyles.Create(nil);

{ the first style }
Style := Styles.Add;
Style.Name := 'Style1';
Style.Font.Name := 'Courier New';

{ the second style }
Style := Styles.Add;
Style.Name := 'Style2';
Style.Font.Name := 'Times New Roman';
Style.Frame.Typ := [ftLeft, ftRight];

{ apply a set to the report }
frxReport1.Styles := Styles;
```

C++:

```
TfrxStyleItem * Style;
TfrxStyles * Styles;

Styles = new TfrxStyles(NULL);

// the first style
Style = Styles->Add();
Style->Name = "Style1";
Style->Font->Name = "Courier New";

// the second style
Style = Styles->Add();
Style->Name = "Style2";
Style->Font->Name = "Times New Roman";
Style->Frame->Typ << ftLeft << ftRight;

// apply a set to the report
frxReport1->Styles = Styles;
```

The set can be created and used in a different way:

Pascal:

```
var
  Style: TfrxStyleItem;
  Styles: TfrxStyles;

Styles := frxReport1.Styles;
Styles.Clear;

{ the first style }
Style := Styles.Add;
```

```
Style.Name := 'Style1';
Style.Font.Name := 'Courier New';

{ the second style }
Style := Styles.Add;
Style.Name := 'Style2';
Style.Font.Name := 'Times New Roman';
Style.Frame.Typ := [ftLeft, ftRight];

{ apply a set to the report }
frxReport1.Styles.Apply;
```

C++:

```
TfrxStyleItem * Style;
TfrxStyles * Styles;

Styles = frxReport1->Styles;
Styles->Clear();

// the first style
Style = Styles->Add();
Style->Name = "Style1";
Style->Font->Name = "Courier New";

// the second style
Style = Styles->Add();
Style->Name = "Style2";
Style->Font->Name = "Times New Roman";
Style->Frame->Typ << ftLeft << ftRight;

// apply a set to the report
frxReport1->Styles->Apply();
```

3.2 Modifying/adding/deleting a style

Modifying a style with a given name:

Pascal:

```
var
  Style: TfrxStyleItem;
  Styles: TfrxStyles;

Styles := frxReport1.Styles;

{ search for a style}
Style := Styles.Find('Style1');

{ modify the font size }
Style.Font.Size := 12;
```

C++:

```
TfrxStyleItem * Style;
TfrxStyles * Styles;

Styles = frxReport1->Styles;
```

```
// search for a style
Style = Styles->Find("Style1");

// modify the font size
Style->Font->Size = 12;
```

Adding a style to the report styles set:

Pascal:

```
var
  Style: TfrxStyleItem;
  Styles: TfrxStyles;

Styles := frxReport1.Styles;

{ add }
Style := Styles.Add;
Style.Name := 'Style3';
```

C++:

```
TfrxStyleItem * Style;
TfrxStyles * Styles;

Styles = frxReport1->Styles;

// add
Style = Styles->Add();
Style->Name = "Style3";
```

Deleting a style with a given name:

Pascal:

```
var
  Style: TfrxStyleItem;
  Styles: TfrxStyles;

Styles := frxReport1.Styles;

{ delete }
Style := Styles.Find('Style3');
Style.Free;
```

C++:

```
TfrxStyleItem * Style;
TfrxStyles * Styles;

Styles = frxReport1->Styles;

// delete
Style = Styles->Find("Style3");
delete Style;
```

After modification the “Apply” method should be called:

```
{ use modifications }
frxReport1.Styles.Apply;
```

3.3 Saving/restoring a set of styles

Pascal:

```
frxReport1.Styles.SaveToFile('c:\1.fs3');
frxReport1.Styles.LoadFromFile('c:\1.fs3');
```

C++:

```
frxReport1->Styles->SaveToFile("c:\\1.fs3");
frxReport1->Styles->LoadFromFile("c:\\1.fs3");
```

3.4 Clearing report styles

Report styles can be cleared in two ways:

```
frxReport1.Styles.Clear;
```

or

```
frxReport1.Styles := nil;
```

3.5 Creating a style library

The following example shows how to create a library and add two sets of styles to it.

Pascal:

```
var
  Styles: TfrxStyles;
  StyleSheet: TfrxStyleSheet;

StyleSheet := TfrxStyleSheet.Create;

{ the first set }
Styles := StyleSheet.Add;
Styles.Name := 'Styles1';
{ here styles can be added to the Styles set }

{ the second set }
Styles := StyleSheet.Add;
Styles.Name := 'Styles2';
{ here styles can be added to the Styles set }
```

C++:

```
TfrxStyles * Styles;
TfrxStyleSheet * StyleSheet;

StyleSheet = new TfrxStyleSheet;
```

```
// the first set
Styles = StyleSheet->Add();
Styles->Name = "Styles1";
//  here styles can be added to the Styles set

// the second set
Styles = StyleSheet->Add();
Styles->Name = "Styles2";
// here styles can be added to the Styles set
```

3.6 Displaying a list of style sets, and application of a selected style

Style libraries are frequently used for displaying available style sets in controls such as “ComboBox” or “ListBox”. The style selected by the user can be applied to the report.

Displaying the list:

```
StyleSheet.GetList(ComboBox1.Items);
```

Using the selected style in the report:

```
frxReport1.Styles := StyleSheet.Items[ComboBox1.ItemIndex];
```

or

```
frxReport1.Styles := StyleSheet.Find[ComboBox1.Text];
```

3.7 Modifying/adding/deleting a styles set

To modify a set with a specific name:

```
var
  Styles: TfrxStyles;
  StyleSheet: TfrxStyleSheet;

{ search for the required set }
Styles := StyleSheet.Find('Styles2');

{ modify a style with the Style1 name from the set found }
with Styles.Find('Style1') do
  Font.Name := 'Arial Black';
```

Adding a set to a library:

```
var
  Styles: TfrxStyles;
  StyleSheet: TfrxStyleSheet;

{ the third set }
Styles := StyleSheet.Add;
Styles.Name := 'Styles3';
```

Deleting a set from a library:

```
var
```

```
  i: Integer;
  StyleSheet: TfrxStyleSheet;

{ search for the third set }
i := StyleSheet.IndexOf('Styles3');
{ if found delete it }
if i <> -1 then
  StyleSheet.Delete(i);
```

3.8 Saving and loading a style library

The file extension for the styles library is “fss” by default.

```
var
  StyleSheet: TfrxStyleSheet;

StyleSheet.SaveToFile('c:\1.fss');
StyleSheet.LoadFromFile('c:\1.fss');
```