Printing in Lazarus

Michaël Van Canneyt

September 24, 2008

Abstract

Many applications need some form of printed reporting. Lazarus has support for printing, quite similar to what Delphi offers. This article shows how to install and use the various printing capabilities of Lazarus

1 Introduction

Printed output is part of most applications. Obviously, these days an application no longer directly sends printer commands to the printer. Instead, each operating system has a printing subsystem, which handles the low-level interface of the printer. The Lazarus LCL has an abstraction layer for this, and this is what should be used if one wishes to print.

On top of the low-level abstraction layer, a reporting engine is built: this allows one to visually design a printed page design, and at run-time, any parameters or data will be filled in and the document can be printed.

Finally, printing support for the IDE must also be installed separately: If this is done, the contents of the Lazarus source editor can also be printed.

All these techniques will be demonstrated.

2 Installation

All the printing functionality is contained in 3 packages:

Printer4lazarus The basic printer support for the LCL. It contains the system specific parts of the printing system.

Printers4lazide This package adds a 'Print' item to the File menu in the IDE, which can be used to print the current file. As of version 0.9.26 of lazarus, a question will be asked if only the selection should be printed.

lazreport This package installs the reporting engine of Lazarus: all reporting components will be installed on the 'Lazreport' tab in the component palette.

The two last packages obviously depend on the first one, so it will be compiled automatically by the IDE.

3 Basic classes

The basic printing object is the `TPrinter` class in the Printers unit, which is part of the LCL. The unit has a `Printer` variable which is instantiated by the the `OSPrinters` unit

(in the Printer4Lazarus package). The `TPrinter` class has a lot of properties and some methods. The main property is a `TCanvas` instance, which represents the page to draw on. The main methods are the following:

SetPrinter Selects a printer. The method is declared as follows:

```
Procedure SetPrinter(aName : String);
```

The sole parameter is the name of the printer that should be selected.

BeginDoc Starts a new print job on the current printer. Many properties are only available after `BeginDoc` was called.

NewPage Starts a new page. This will save the current page, and clears the canvas for a new page to be drawn. The Pagenumber property is also augmented with one.

EndDoc Ends the print job, and actually sends the job to the printer.

The last three methods do not accept parameters.

The main properties needed to print something are the following:

Canvas This canvas represents the current page.

PaperSize A class describing the current paper size. Its `Papername` property can be set to one of the many common names to set the paper size, such as `A4` or `Letter`.

Orientation This can be set to `poPortrait` or `poLandScape` to set the paper orientation.

Copies Contains the number of copies that should be printed.

PageHeight The height of the page in pixels (dots), read-only.

PageWidth The width of the page in pixels (dots), read-only.

PageNumber The current page number, read-only.

Title The print job title, which will show up in the printer queue dialog.

XDPI The horizontal resolution in DPI (Dots Per Inch), read-only.

YDPI The vertical resolution in DPI, read-only.

To select a printer, the `PrinterDlgs` can be used to create a `TPrinterDialog` instance. This will show a printer selection dialog, and will allow to set the number of copies, the paper size and orientation. When the dialog quits, the properties of the standard `Printer` instance will reflect what the user has selected in the dialog.

The canvas object is not different from the canvas used in GUI components to draw themselves on the screen and has been treated in a previous contribution in this magazine. This makes it particularly easy to share the same drawing code between the printing and screen drawing code.

4 Basic usage

To demonstrate how to use the printer in Lazarus, a small component will be developed that can print a directory listing. To show that printing to a page is not different from printing to a screen, the same routine that is used to print the listing will be used to draw the listing on the screen - for a preview this is a convenient property of the printing process.

The component has the following (simplified) declaration:

```
TDirectoryLister = Class(TComponent)
Public
  Procedure DrawPage(ACanvas : TCanvas;
                     APageNo,ALinesPerPage,ADPI : Integer);
  Procedure ReadDirectory;
  Procedure Execute;
  Function GetLinesPerPage(ACanvas : TCanvas;
                           ACanvasHeight,
                           ADPI: Integer): Integer;
  Function Points(AUnits: Double; ADPI: Integer): Integer;
  Property Entries : TDirectoryEntries;
Published
  Property Directory : String;
  Property Font : TFont;
  Property Title : String;
  Property Options : TListOptions;
  Property TopMargin : Double;
  Property BottomMargin : Double;
  Property LeftMargin : Double;
  Property RighMargin : Double;
  Property LineSpacing : Integer;
  Property Columns : TDirListColumns Read FColumns Write SetColumns;
end;
```

The meaning of most of these methods and properties should be intuitively clear, with the `Directory` property as the most important : it determines of which directory the contents should be printed. The `Entries` property is filled by the `ReadDirectory` with `TDirectoryEntry` collection items, one for each file in the directory. The measurements for the margins are in centimeters, while the linespacing is in dots. The list options is a set of the following values:

```
TListOption = (loHidden,loDirectory,loSymlinks,
               loNumbered,loDrawHeader,loSorted);
```

The first three values determine what special files should be shown, and the last three determine whether line numbers should be drawn, whether a header should be drawn above the columns and lastly whether the directory contents should be printed in an alphabetically sorted order.

Once all properties are set, the `Execute` method will do some of the actual work. It is dissappointingly simple:

```
procedure TDirectoryLister.Execute;

var
  P,PageCount,LinesPerPage : integer;
```

```
  LineHeight : Double;

begin
  If Not DoPrinterSetup then Exit;
  Printer.Title := 'Directory Listing:'+Directory;
  Printer.BeginDoc;
  Printer.Canvas.Font := FFont;
  ReadDirectory;
  LinesPerPage:=GetLinesPerPage(Printer.Canvas,
                                Printer.PageHeight,
                                Printer.YDPI);
  PageCount := FEntries.Count div LinesPerPage;
  if FEntries.Count mod LinesPerPage <> 0 then
    Inc(PageCount);
  try
    for P := 1 to PageCount do
      begin
      DrawPage(Printer.Canvas,P,LinesPerPage,Printer.YDPI);
      if P<>PageCount then
        Printer.NewPage;
      end;
    Printer.EndDoc;
  except
    on E:Exception do
      begin
      Printer.Abort;
      raise;
      end;
  end;
end;
```

It starts by calling the printer setup dialog, and if it was cancelled, the routine exits at once. After that the printjob title is set, and a new document is started. The font is set, and the directory contents are read in the `ReadDirectory` method.

To know how many pages must be printed, the `GetLinesPerPage` method is called: it calculates the number of lines per page, depending on the page height and resolution. Once the number of lines per page is known, the number of needed pages is calculated, and a loop is started which prints each page: after each page is drawn, the `NewPage` method is called to start the next page. Finally, the `EndDoc` method is called to send the job to the printer. If an error is detected, then the `Abort` method of the printer is used to cancel the current job.

To calculate the number of lines that can be drawn, the `GetLinesPerPage` routine is used:

```
function GetLinesPerPage(ACanvas: TCanvas;
                         ACanvasHeight,
                         ADPI : Integer): Integer;

Var
  H : Integer;
  DPC : Integer;

begin
```

```
  DPC:=Round(ADPI/AnInch);
  H := ACanvas.TextHeight('X') + LineSpacing;
  Result:=Round((ACanvasHeight-DPC*(FTopMargin-FBottomMargin))/H-3);
  If FDrawHeader then
    Dec(Result,2);
end;
```

The code shows why the actual canvas is needed to determine the number of lines: the `TextHeight` method of `TCanvas` is used to determine the height of one line. The rest is simple mathemathics: calculating ratios. The `AnInch` constant (2.54) is used to convert centimeters to inches. If a header is requested, then the number of lines is decreased with 2: one for the header text, one to draw a line under the header at half line height.

This kind of routine will be encountered often when printing, as measurements in centimeters or inches need to be converted to dots for exact drawing.

The `DrawPage` routine takes care of the actual drawing, but before this method is examined, a word should be said on the `Columns` property. This is a collection of `TDirListColumnItem` items, defined as follows:

```
TDirListContent = (dcLineNumber,dcName,dcTimeStamp,
                   dcDate,dcTime,dcAttributes);

TDirListColumnItem = Class (TCollectionItem)
public
  Function DisplayString(E : TDirectoryEntry) : String;
Published
  Property Width : Double;
  Property Content : TDirListContent;
  Property TimeStampFormat : String;
  Property Title : String;
end;
```

The `Columns` property of `TDirectoryLister` contains an item for each column that should be drawn in the listing: the `Content` property determines what should be drawn in the column. The title and width properties speak for themselves (width in centimeters) and the `TimeStampFormat` is used to format the datetime/date/time columns.

The `DisplayString` function will - starting from a `TDirectoryEntry` instance - create a string to be displayed. This string can be used in the drawing routine. The drawing routine does not need to know how this functions in order to draw the listing.

Now the `DrawPage` method can be examined:

```
procedure DrawPage(ACanvas : TCanvas;
                   APageNo, ALinesPerPage, ADPI : Integer);

var
  I,J,Min,Max,H,
  FMarginX,FCurrentX,FCurrentY : integer;
  W : Double;
  s: string;
  LineNum: integer;
  E : TDirectoryEntry;
  CC : Boolean;

begin
```

```
  CC:=(FColumns.Count=0);
  if cc then
    CreateDefaultColumns;
  Try
    Min:=(Pred(APageNo)*ALinesPerPage)+1;
    Max:=(APageNo*ALinesPerPage);
    If (Max>=FEntries.Count) then
      Max:=FEntries.Count-1;
    FCurrentY:=Points(FTopMargin,ADPI);
    FMarginX:=Points(FLeftMargin,ADPI);
    H:=ACanvas.TextHeight('X')+FLineSpacing;
    If FDrawHeader then
      begin
      FCurrentX:=FMarginX;
      For J:=0 to FColumns.Count-1 do
        begin
        S:=FColumns[J].Title;
        ACanvas.TextOut(FCurrentX,FCurrentY,S);
        W:=FColumns[J].Width;
        ACanvas.Line(FCurrentX,
                     FCurrentY+Round(H*1.5),
                     FCurrentX+Points(W,ADPI),
                     FCurrentY+Round(H*1.5));
        FCurrentX:=FCurrentX+Points(W,ADPI);
        end;
      Inc(FCurrentY,H*2);
      end;
    for I:=Min to Max do
      begin
      FCurrentX:=FMarginX;
      E:=FEntries[I-1];
      For J:=0 to FColumns.Count-1 do
        begin
        S:=FColumns[J].DisplayString(E);
        ACanvas.TextOut(FCurrentX,FCurrentY,S);
        FCurrentX:=FCurrentX+Points(FColumns[J].Width,ADPI);
        end;
      Inc(FCurrentY,H);
      end;
  Finally
    If CC then
      FColumns.Clear;
  end;
end;
```

If no columns are defined, the routine starts by defining a set of default columns. After that, the range of entries is determined from the page number, and the starting point on the page is determined from the margins. If a header was requested, a header is drawn: this simply loops over the items in the columns property, and draws the title for each item, together with a line: it updates the X and Y positions as it goes.

After the header is drawn, the real work starts, with a loop over the entries that should be drawn on the current page. Each entry is again printed using a loop over the columns: the text to be drawn for each column is fetched using the `DisplayString` method of the

column. Again the current X and Y positions are updated as the loops are executed.

The method ends by destroying the default columns if needed.

Note that the `DrawPage` did not use any printer-specific methods: it uses just the methods of the canvas: all other information it needs (DPI, lines per page) it got from the caller.

This makes the method also suitable for drawing the directory listing on a screen canvas: this can be used for a preview. To demonstrate this, a demonstration program is made that allows to select a directory (using a standard `TDirectoryEdit` component), and which has 2 buttons: `BPrint` and `BDraw`. It further has a `TPanel` instance (`PListing`), which will be used to draw the preview in.

The `BPrint` button has an extremely simple `OnClick` handler:

```
procedure TMainForm.BPrintClick(Sender: TObject);
begin
  FLister.Directory:=DEDir.Directory;
  FLister.Execute;
end;
```

The `FLister` variable is an instance of the `TDirectoryLister` component.

The `BDraw` button's `OnClick` handler is equally simple:

```
procedure TMainForm.BDrawClick(Sender: TObject);
begin
  FLister.Directory:=DEDir.Directory;
  FLister.ReadDirectory;
  DrawPage(PListing.Canvas,PListing.ClientRect);
end;
```

It calls the `ReadDirectoy` method, and then calls the form's `DrawPage` method, passing it the canvas of the `PListing` panel, with the `ClientRect` of the panel for the available space. The `DrawPage` is also simple:

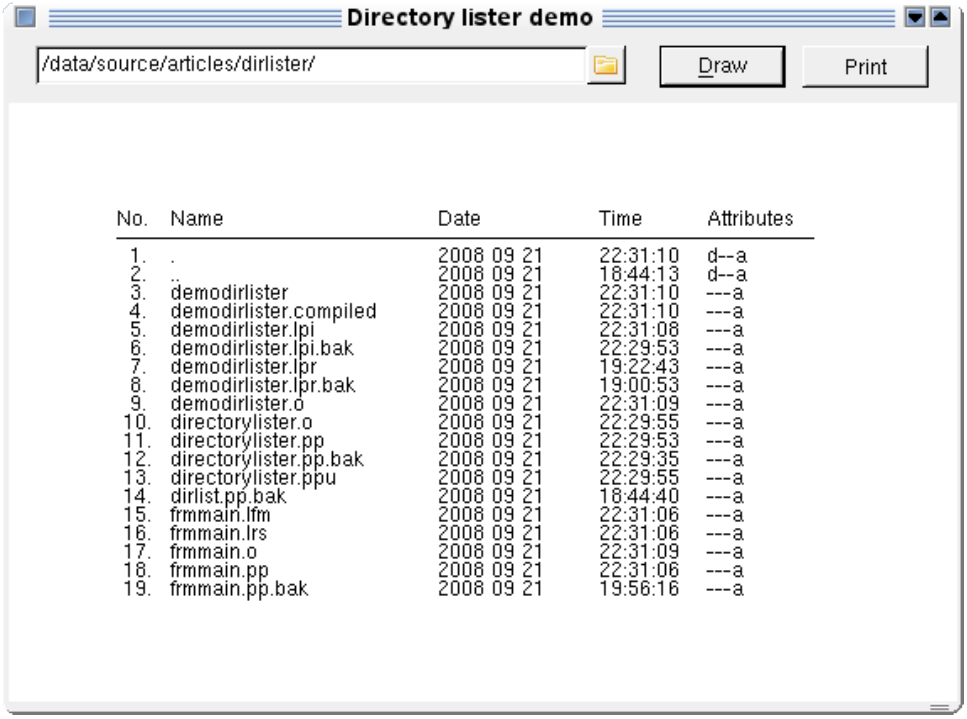
```
procedure TMainForm.DrawPage(ACanvas : TCanvas; ARect : TRect);

Var
  LPP : Integer;

begin
  ACanvas.Brush.Color:=clWindow;
  ACanvas.FillRect(ARect);
  LPP:=Flister.GetLinesPerPage(ACanvas,
                               ARect.Bottom-ARect.Top,
                               Screen.PixelsPerInch);
  FLister.DrawPage(ACanvas,1,LPP,Screen.PixelsPerInch);
end;
```

This looks very much like the `Execute` method of the `TDirectoryLister` component, with the exception that first, an empty rectangle is drawn on the panel. After that, the number of lines per page is calculated - using the DPI of the screen - and the `DrawPage` method of the `TDirectoryLister` is called. The result can be seen in figure 1 on page 8 The routines for actually reading the directory listing and creating the display string have not been shown: they are not relevant for the understanding of the printing, but the interested reader can look up the routines in the source code accompagnying this issue.

Figure 1: Directory lister in action



5 Reporting

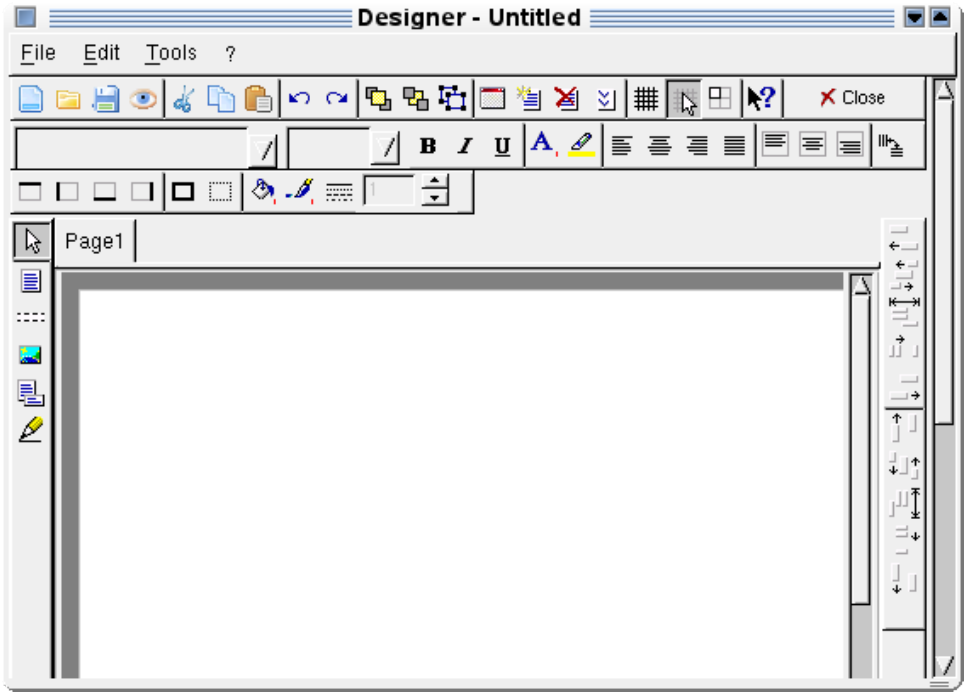
The above has shown that there is nothing difficult about printing on a page: it is no different from drawing on a screen. It can of course get tiresome, and if a lot of different data is to be printed, then an easier method is required: the above example was particularly easy because it contains only columnar data: if more elaborate layouts are needed, then it soon becomes quite cumbersome.

Fortunately, Lazarus comes with Lazreport, a reporting engine: it is a port of the FreeReport engine (an early FastReport version) to Lazarus. This is a banded report designer, which allows to design a report visually at design time (or at runtime, for that matter), and display and print the finished report at runtime.

Although primarily oriented towards printing data from databases (using `TDataset` descendents), it can also be used to print any other data one wishes to feed it. The basic concept is the same:

- The report consists of one or more designed pages.
- Attached to the report are one or more sources of data. This can be TDataset data, but also user-generated data.
- Each page contains one or more 'bands': some bands (e.g. the report title) are printed only once, others once per page (e.g. page title and footer) and other bands are printed once for each data record that the data source provides.
- On each band in the report, one or more visual elements are laid out: they are printed each time the band is printed, in exactly the same position relative to the band's top-left corner. Visual elements can be plain text (or formulae), images, lines or other graphic objects.

Figure 2: The report designer



All this can be designed visually, without a single line of code.

A complete description of a reporting engine is beyond the scope of this article, but nevertheless 2 examples will be given to demonstrate how it can be used to print simple reports.

For the first example, the address book demonstration program in lazarus will be enhanced so it can print a listing of the addresses in the book. To do this, first the `Lazreport` package must be installed. When this is done, the `LazReport` tab appears on the component palette.

From the component palette, 2 components should be dropped on the main form of the `addrbook` project: a `TFRReport` component, and a `TFRDBDataset` component. The first will contain the report definition. The report definition can be stored in a separate file, but it can also be stored in the lazarus form file: this is easier, since then no additional files must be distributed with the application. For this, the `StoreInDFM` property must be set to `True`. The `TDRDBDataset` component must be connected through it's `Dataset` property to the `DBA` dataset. The `DBA` dataset must be set to `Active` to be able to design the report.

To design the report, the component editor must be used: click right on the report instance, and select 'Design Report'. The report designer appears as in figure 2 on page 9. The top area of the report designer looks quite like a word processor: it contains the usual elements to set the color, font and frame around a text, as well as load save and preview buttons. The right area contains 2 toolbars, familiar from any GUI designer: to align report elements vertically and horizontally. Most of these should be quite familiar, all buttons have tooltips to explain their function.

The left area is the equivalent of the component palette in the Lazarus IDE. Clicking one of these buttons allows to drop a new report element. The default buttons allow to drop the following elements in a report:

- A text element. The text can contain formulae, variables and references to database

fields.

- A band. Once the band has been dropped on the form, the type of the band must be chosen.
- An image.
- A subreport: Reports can be nested. The subreport will appear on a new tab, and can be designed separately.
- A line: various lines can be drawn.

More can be added, but these buttons are standard.

To design an address listing, 2 bands can be dropped on the report: the first is a page header. This band will be printed on the top of each page. It contains a title, and a column header for each column of data. To create this band, the band button must be clicked, and then a click anywhere in the report page is sufficient to drop a new band on the form: once it was dropped, the type of the band must be chosen.

To design the page title and column titles, the text element (the first) button must be clicked, and then a click somewhere in the page header band will create the new element. A memo editor appears in which the desired text can be typed. Repeating this process a couple of times is sufficient to create all column headers.

To print the address records of the database, a 'Master Data' band must be dropped on the report page. Once it is dropped, the report designer will ask which dataset should be used for this master data report, and will present a list of available datasets - obviously the 'DBA' dataset should be chosen. On this band, a text element must be dropped for each column that should be printed. The first column should contain e.g. the following text:

```
[DBA."FIRSTNAME"] [DBA."LASTNAME"]
```

The meaning of this text is not hard to guess: everything between square brackets `[ ]` is interpreted as a formula, which the report engine will try to evaluate when the element must be printed, and the result will be inserted in the text at the location of the formula.

The formula `DBA.¨FIRSTNAME¨` means that the value of the field `FIRSTNAME` from dataset `DBA` should be inserted. A second formula with a second field (the last name) is separated from the first formula by a space: the space will be inserted verbatim in the text. The result of all this is that the first name will be followed by the last name, separated by a space.

Special formulas can be made, for instance

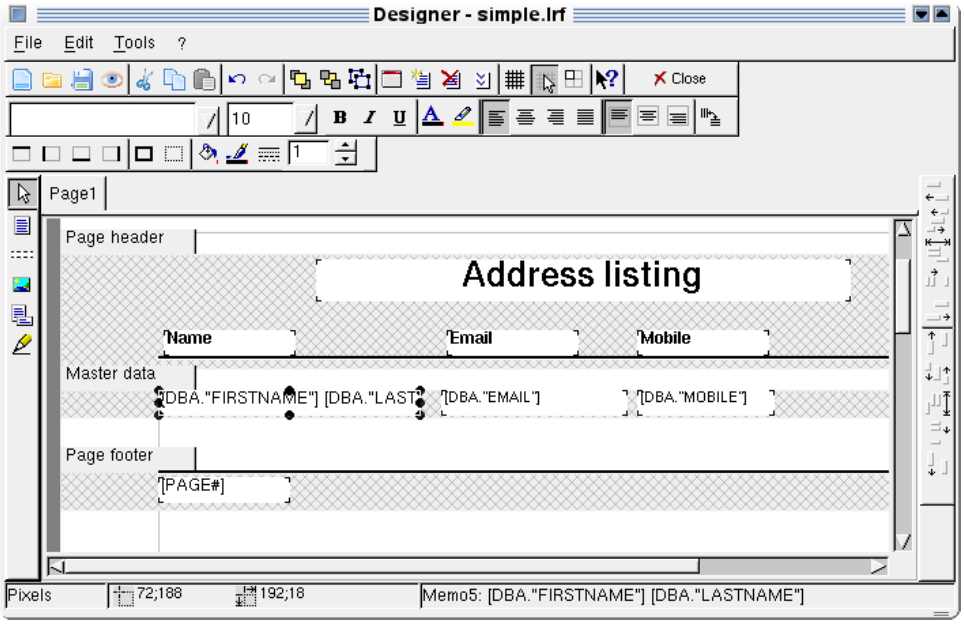
```
[PAGE#]
```

can be used to insert a page number (in fact, a page footer band is inserted in the report with this formula on it). The result of all this should look like figure 3 on page 11. The finished report can be printed or shown on the screen with a few lines of code. To do this, 2 menu items are added to the 'File' menu: one to preview, one to print the report:

```
procedure TMainForm.MIPreviewClick(Sender: TObject);
begin
  If FRReport1.PrepareReport then
    FRReport1.ShowPreparedReport;
end;

procedure TMainForm.MenuItem1Click(Sender: TObject);
```

Figure 3: The finished report




```
begin
  If FRReport1.PrepareReport then
    FRReport1.PrintPreparedReport('',1);
end;
```

Both event handlers start by calling `PrepareReport`: this will build the report in memory, and will return `True` if the report was prepared succesfully. The `ShowPreparedReport` will show a preview window which can be used to preview the report before printing. The `PrintPreparedReport` call takes 2 arguments: a string describing the pages to be printed, and a number of copies to be printed. An empty string means that all pages should be printed. An example of a more complicated string is

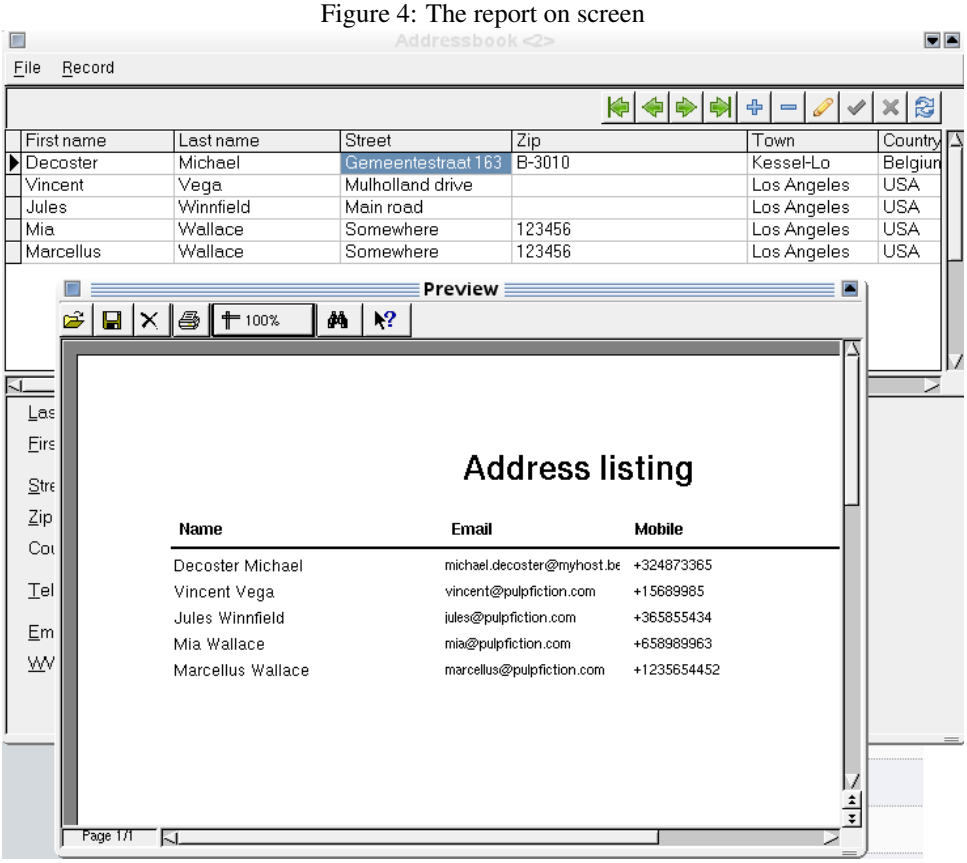
```
1,2,6-10
```

This will print pages 1,2 and 6 till 10. Pressing the 'Preview' menu will result in the report shown in figure 4 on page 12. It is possible to let the user edit the report. Allowing this has the advantage that small changes to a report (adding a logo, changing the font) can be done by the end user, and does not need to be programmed by the programmer. To do this, a `TFrDesigner` component must be dropped on the main form of the address book example, and a 'Design' menu item can be added to the menu. The `OnClick` handler of this menu item is as simple as can be:

```
procedure TMainForm.MIDesignClick(Sender: TObject);
begin
  FRReport1.DesignReport;
end;
```

The user can design and save the report to file. To print the changed report, the changed file must be loaded from file and then printed - which is automatic if the report stored as file and not in the .DFM file.

Figure 4: The report on screen



6 Custom reporting

Despite the fact that the reporting engine is geared towards printing data from datasets, it is perfectly suitable for printing any data one wishes to feed to it: it can be perfectly used to print a directory listing, as in the first example.

To show this, we'll demonstrate how to print the contents of a listview control that shows the contents of a directory. The example program uses the same `TDirectoryEntries` collection as the directory listing: an instance of this collection is created when the main form is created:

```
procedure TMainForm.FormCreate(Sender: TObject);
begin
  FEntries:=TDirectoryEntries.Create(TDirectoryEntry);
  DEMain.Directory:=ExtractFilePath(Paramstr(0));
end;
```

On the form, a `TDirectoryEdit` and a TButton are dropped, as well as some checkboxes to determine whether hidden files and directories should be shown as well. In the `OnClick` handler of the button, the following code is executed:

```
procedure TMainForm.BreadClick(Sender: TObject);
begin
  FCurrentDirectory:=DEMain.Directory;
  ReadDirectory;
  ShowDirectory;
end;
```

The `ReadDirectory` is the same routine as used by the `TDirectoryLister` component, and will not be shown here. The `ShowDirectory` copies the contents of the collection to the listview:

```
procedure TMainForm.ShowDirectory;

Var
  L : TListItem;
  I : integer;
  S : String;

begin
  With LVDirectory do
    begin
    BeginUpdate;
    try
      Items.Clear;
      For I:=0 to FEntries.Count-1 do
        begin
        L:=LVDirectory.Items.Add;
        L.Caption:=FEntries[i].FileName;
        S:=FormatDateTime('yyyy/mm/dd',FEntries[I].FTimeStamp);
        L.SubItems.Add(S);
        S:=FormatDateTime('hh:nn:ss',FEntries[I].FTimeStamp);
        L.SubItems.Add(S);
        L.SubItems.Add(FEntries[I].AttributeString);
        L.Data:=FEntries[i];
```

```
        end;
    finally
      EndUpdate;
    end;
    end;
end;
```

There is little mysterious about this code, it just loops over the items in the collection and copies the data to the listview. As can be seen, the listview has 4 columns: one for the filename, file date, file time and attributes.

To print the contents of the listview, a `TFRUserDataset` is used. It has 3 events, which it uses to emulate a loop (through an imaginary dataset).

OnFirst Is called to position the loop variable on the initial value (or the dataset on the first record).

OnNext Is called to position the loop variable on the next value (or the dataset on the next record).

OnCheckEOF Is called to see if the end of the loop is reached. This is called quite often - in fact, twice as much as `OnNext` is called.

This can be used to emulate a dataset from the listview: a loop variable is introduced, which denotes the current item in the list. In the 3 events, the following code is executed:

```
procedure TMainForm.DirDataFirst(Sender: TObject);
begin
  FCurrentItem:=0;
end;

procedure TMainForm.DirDataNext(Sender: TObject);
begin
  Inc(FCurrentItem);
end;

procedure TMainForm.DirDataCheckEOF(Sender: TObject; var Eof: Boolean);
begin
  EOF:=(FCurrentItem>=LVDirectory.Items.Count-1);
end;
```

It should be quite clear that this emulates a loop over all items in the listview.

To retrieve the data from the listview, the `OnGetValue` event handler of `TFRReport` is used. This event handler is called whenever the reporting engine encounters a variable or data field that it does not know or find in the attached datasets. In the case of the directory listing, this can be used to introduce 5 variables: `DirectoryName`, `FileName`, `FileDate`, `FileTime` and `FileAttributes`. They correspond to the 4 columns of the list view, and the name of the current directory. The `OnGetValue` event handler can then be coded as follows:

```
procedure TMainForm.FRDirListGetValue(const ParName: String;
  var ParValue: Variant);

Var
  L : TListItem;
```

```
begin
  L:=LVDirectory.Items[FCurrentItem];
  If (ParName='DirectoryName') then
    ParValue:=FCurrentDirectory
  else if (ParName='FileName') then
    Parvalue:=L.Caption
  else if (ParName='FileDate') then
    Parvalue:=L.SubItems[0]
  else if (ParName='FileTime') then
    ParValue:=L.SubItems[1]
  else if (ParName='FileAttributes') then
    ParValue:=L.SubItems[2];
end;
```

First, the 'Current record' is retrieved from the ListView component, using the `FCurrentItem` variable. After that, the name of the requested variable is examined, and the correct 'field value' is returned.

All that is left to do is to design the report. This is quite trivial. For each of the 4 columns in the report, a formula of the form

```
[FileName]
```

is used - obviously 'Filename' must be replaced with the correct value for each column. The page title formula is something like

```
Directory listing of [DirectoryName]
```

Adding some column titles is trivial, and the result of all this code can be seen in figure 5 on page 16. Obviously, this report could haven been created without the use of the `TListView`, directly from the `TDirectoryEntries` collection. This is a trivial change, and is left as an exercise for the reader.

Note that since there is no real data available at design time, it is not possible to preview the report in the IDE. This can be remedied by designing the report in the running application, save it to disk and then load the saved file in the report in the IDE.

7 Conclusion

Hopefully, this article has shown that it is easy to print documents in Lazarus. For very simple layouts that need little change, the document can be easily drawn on the printer canvas. For more complex layouts, or prints that the user may want to change, the use of the `LazReport` reporting engine is recommended, even if no datasets are used in the application.

Figure 5: The directory list report in action