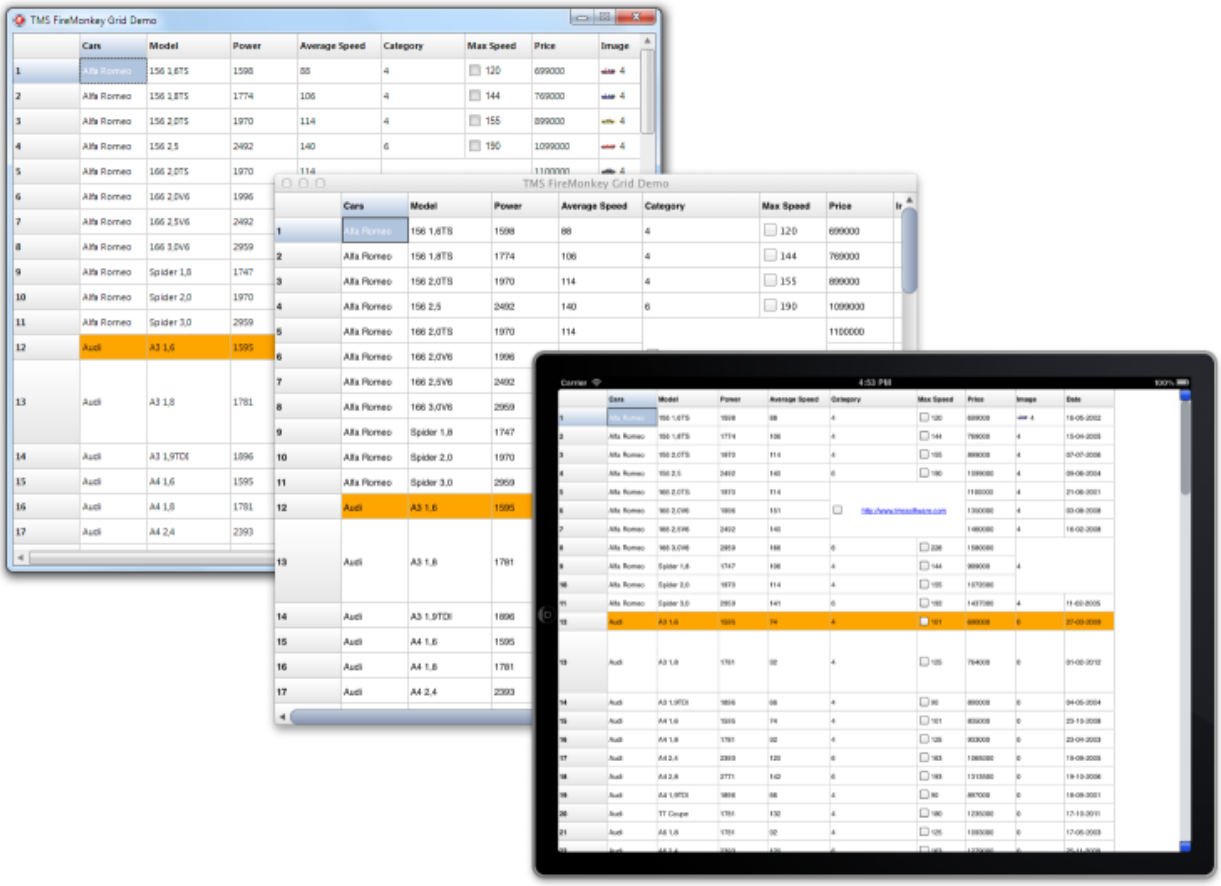
# TMS Grid for FireMonkey Whitepaper

**September 2012**

**Web: http://www.tmssoftware.com**
**Email: info@tmssoftware.com**

# Introducing TTMSFMXGrid: a flexible, productivity feature-packed cross platform grid for FireMonkey



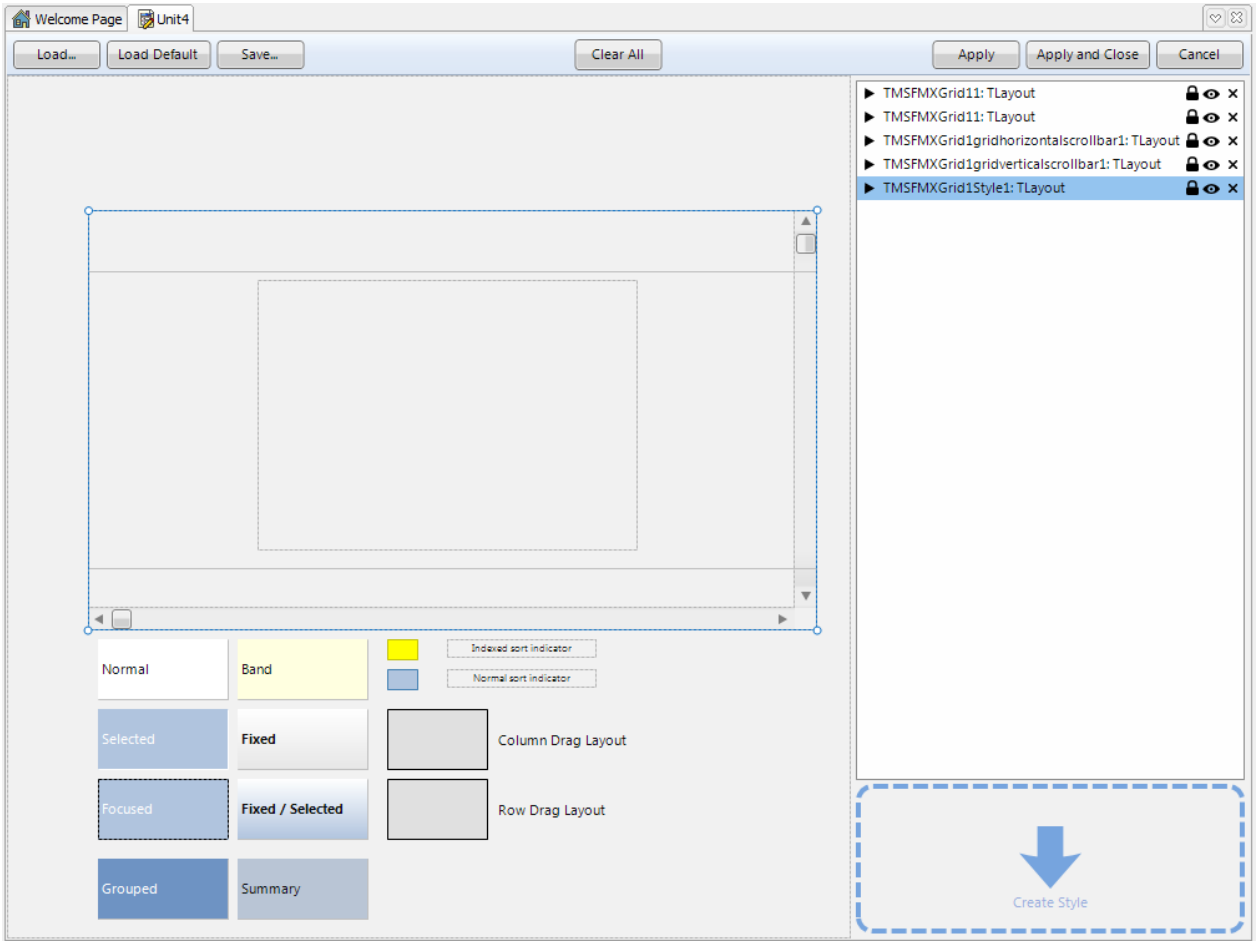

## Introduction

As soon as Delphi XE2 with the new FireMonkey framework was released, a feature-rich and high performance grid was one of the most requested components for this new framework. While we had already worked on a grid for the VCL framework for over 15 years now, it immediately became clear that first of all, there was not a quick way to port this VCL grid to the FireMonkey framework and secondly, that is was also highly undesirable to do such port. We were convinced that a TMS grid for FireMonkey only made sense as a long term project when it was from the ground up designed in the spirit and according to the principles of the FireMonkey framework, being style-able and fully cross platform a fundamental requirement.
As the FireMonkey framework is fundamentally different from the VCL framework, the TMS team first wet its feet with the development of several more lightweight FireMonkey components. Only after we felt sufficiently experienced and introduced to the inner workings of the FireMonkey framework, we started the architecting of a new grid. A solid house is built on strong foundations.

## Requirements

When we met to set the requirements of the new grid, this was the shortlist:

1. Being fully cross platform: the grid had to work well on Windows 32bit, 64bit, Mac OS-X, iOS and possibly also Android & Linux in the future. This means relying solely on the FireMonkey framework and only when strictly needed provide wrappers for OS specific functions to offer the same functionality across all platforms.
2. Fully respect the principles of the FireMonkey framework thus being style-able. The different visual elements of the grid need not only to be editable in the IDE style editor to modify fill, font, strokes, … but also offer the flexibility of full customization through modifying the style template.
3. High performance: we wanted a grid that can display 500.000 rows and 10.000 columns without any significant performance hit. This is perhaps the most difficult requirement when designing something with the FireMonkey framework, especially when deploying to iOS. Weeks were spent on finding the right architecture with the right balance between performance and customizability.
4. Feature-rich, data oriented: in the first place, the grid is oriented towards LOBs (line of business applications). Displaying huge amounts of data, allowing editing with various inplace editor types, searching, filtering, sorting, importing & exporting to various formats, printing, …
5. Familiar with VCL TAdvStringGrid: The VCL TAdvStringGrid has a history of over 15 years by now. This means that many of its features were developed over time. Through all these years, keeping backwards compatibility was a top priority and that led here and there to suboptimal organization of properties, events and method interfaces. Interface compatibility with TAdvStringGrid was as such not a goal of the TMS Grid for FireMonkey. The grid is designed partly in the spirit of TAdvStringGrid so that users will be quickly familiar with it but interface organization was cleaned up to make it more intuitive for new users or users with no experience with the VCL TAdvStringGrid.
6. Work with LiveBindings and in particular Visual LiveBindings in XE3.




*Being style-able as core feature: TMS Grid for FireMonkey in the IDE style editor*

## Architecture of the TMS Grid for FireMonkey

The grid consists of a data layer and a UI layer. The data layer takes care of memory management, organization, manipulations like sorting, grouping, filtering of cell data, i.e. cell content and cell properties. The UI layer handles the display of FireMonkey objects that represent the grid and takes care of dealing with all mouse & keyboard events.
In its basic layout, a grid is a matrix of cells with mainly fixed cells (not editable) and normal cells. The base cell FireMonkey object is TTMSFMXGridCell. A fixed cell will not scroll along with normal cells and thus remain visible on any of the 4 sides of the grid. This number of fixed rows and/or columns on the 4 sides of the grid is controlled by properties: grid.FixedRows, grid.FixedColumns, grid.FixedFooterRows, grid.FixedRightColumns. In addition to fixed, non-scrolling rows and/or columns, the grid can also perform column freezing. These are columns or rows of normal cells that will not scroll along with the other columns or rows in the grid. The number of freeze columns and rows is set with grid.FreezeColumns, grid.FreezeRows. Cells are accessible via grid.Cells[Column,Row]:string and the selected cell(s) can be set with properties:

```
grid.Selection := CellRange(StartCol,StartRow,EndCol,EndRow);
```

```
grid.FocusedCell := Cell(Col,Row);
```

The grid features several selection modes: single cell selection, single row selection, single column selection, cell range selection, row range selection, column range selection, disjunct row selection, disjunct cell selection and disjunct column selection. The selection mode is chosen with the property:

```
grid.SelectionMode: TTMSFMXGridSelectionMode;
```

The scroll position in the grid can be programmatically set or retrieved via the properties grid.LeftCol: integer, grid.TopRow: integer.
Note that scrolling in the grid can be performed in two ways: cell scrolling and pixel level scrolling. In cell scrolling mode, the minimum quantity of a scroll is an entire column or row, in pixel scrolling mode, scrolling is per pixel and can thus be done on sub cell level. The scrolling mode is controlled by the property:

```
grid.ScrollMode = (smCellScrolling, smPixelScrolling)
```

The size of columns & rows is controlled by grid.ColumnWidths[ColumnIndex], grid.RowHeights[RowIndex] and it can be configured that the user can resize columns or rows at runtime with: grid.Options.ColumnSizing,grid.Options.FixedColumnSizing, grid.Options.RowSizing, grid.Options.FixedRowSizing.

## Importing and exporting data with the TMS Grid for FireMonkey

The grid is able to import data from different file formats as well as export it to different formats. The top left cell from where the loading of data starts is set with grid.IOOffset: TPoint. Typically this is set to the first normal cell in the grid, i.e.

```
grid.IOOffset := Point(grid.FixedColumns,grid.FixedRows);
```

Following formats and methods are available:

1. Simple format to persist cell text, column widths, row heights (file format compatible with VCL TAdvStringGrid)

   ```
   grid.LoadFromFile(FileName: string);
   grid.SaveToFile(FileName: string);
   ```

2. CSV files

   ```
   grid.LoadFromCSV(FileName: string);
   grid.SaveToCSV(FileName: string);
   ```

3. Fixed column width text files

   ```
   grid.LoadFromFixed(FileName: string);
   ```

```
grid.SaveToFixed(FileName: string);
```

4. Memory stream with cell text, column widths, row heights

```
grid.LoadFromStream();
grid.SaveToStream();
```

5. Microsoft .XLS files

   The component TTMSFMXGridExcelIO is offered that allows to import and export to Microsoft .XLS file format.

6. Export to RTF files

   The component TTMSFMXGridRTFIO is offered that allows to export the grid to RTF file format.

7. Export to XML files

   grid.SaveToXML() performs an export of the data in XML format.

## Sorting, grouping and filtering data in the grid

| | Type | CC | Hp | Cyl | Kw | Price |
|---|---|---|---|---|---|---|
| ⊟ | Alfa Romeo | | | | | |
| | 156 1,6TS | 1598 | 88 | 4 | 120 | 699000 |
| | Spider 1,8 | 1747 | 106 | 4 | 144 | 999000 |
| | 156 1,8TS | 1774 | 106 | 4 | 144 | 769000 |
| | 156 2,0TS | 1970 | 114 | 4 | 155 | 899000 |
| | Spider 2,0 | 1970 | 114 | 4 | 155 | 1072000 |
| | 166 2,0TS | 1970 | 114 | 4 | 155 | 1100000 |
| | 166 2,0V6 | 1996 | 151 | 6 | 190 | 1350000 |
| | 156 2,5 | 2492 | 140 | 6 | 190 | 1099000 |
| | 166 2,5V6 | 2492 | 140 | 6 | 190 | 1460000 |
| | Spider 3,0 | 2959 | 141 | 6 | 192 | 1437000 |
| | 166 3,0V6 | 2959 | 166 | 6 | 226 | 1580000 |
| | | | | | 169.1818181 | 12464000 |
| ⊟ | Audi | | | | | |
| | A3 1,6 | 1595 | 74 | 4 | 101 | 690000 |
| | A4 1,6 | 1595 | 74 | 4 | 101 | 835000 |
| | A3 1,8 | 1781 | 92 | 4 | 125 | 764000 |

*Grid grouped by column 1 and with nodes to expand/collaps groups*

The grid has built-in sorting capabilities as well as filtering capabilities. Sorting can be performed from the UI by clicking the column header to perform a sort on a column and can be done programmatically as well. Four types of sorting are available:

Single column sort
Multi column sort
Single column grouped sort
Multi column grouped sort

For single column sort, the sort is based on just a single column clicked. Clicking a new column will perform the sort on the new column clicked. In multi column sort mode, additional columns can be used as sort criteria by performing a shift-click on the column header. When the grid is grouped, the equivalent sorting capabilities are available but limited to groups, i.e. rows are sorted within groups rather than within the full grid.
Programmatically sorting can be done with:

```
  grid.SortData(ColumnIndex, SortDirection);  // single column sort

 grid.SortIndexes.AddIndex(ColumnIndex1, SortDirection1);  // multi
column sort on column 1,2
  grid.SortIndexes.AddIndex(ColumnIndex2, SortDirection2);
  grid.SortIndexed;
```

Note that to control sorting from the UI, the events OnCanSortColumn, OnColumnSorted events are available.

To perform grouping of data on a column, simply call

```
  grid.Group(ColumnIndex)
```

and to undo the grouping, call:

```
  grid.UnGroup;
```

Filtering can be performed programmatically. The filter conditions are added via the grid.Filter collection property and filtering is started by calling grid.ApplyFilter;

```
 var
    fltr: TFilterData;

  grid.Filter.Clear;
  fltr := grid.Filter.Add;
  fltr.Condition := 'Some condition';
  fltr.Column := ColumnIndex1;

  fltr := grid.Filter.Add;
  fltr.Condition := 'Other condition';
  fltr.Column := ColumnIndex2;
  fltr.Operation := foAND;
```

```
  grid.ApplyFilter;
```

## Cell properties and controls

In its most basic form, two types of cells exist, the fixed cells and normal cells. A normal cell can be in normal state, in selected state and in focused state. The style for these different states can be edited in the IDE via the style editor but can be programmatically accessed as well. To programmatically change the default style for a normal cell to make it appear with a yellow background color and red border, following code could be used:

```
  grid.GetDefaultNormalLayout.Layout.Fill.Color := claYellow;
  grid.GetDefaultNormalLayout.Layout.Stroke.Color := claRed;
  grid.ApplyStyle;
```

The style of cells can also be dynamically changed with the event OnGetCellLayout. This example implementation of the OnGetCellLayout event will set the font color of cells with a value higher than 50 to red and as this is a column with numbers only, set its alignment to right justified:

```
procedure TForm1.TMSFMXGrid1GetCellLayout(Sender: TObject; ACol, ARow:
Integer;
  ALayout: TTMSFMXGridCellLayout; ACellState: TCellState);
var
  i,e: integer;
begin
  if (ACol = 2) and (ARow >= TMSFMXGrid1.FixedRows) then
  begin
    ALayout.TextAlign := TTextAlign.taTrailing;
    val(TMSFMXGrid1.Cells[ACol, ARow],i,e);
    if i > 50 then
      ALayout.FontFill.Color := claRed;
  end;
end;
```

Programmatic access to many cell appearance characteristics are also available:

```
grid.Alignments[ColumnIndex, RowIndex]: TTextAlign;
grid.Colors[ColumnIndex, RowIndex]: TAlphaColor;
grid.FontColors[ColumnIndex, RowIndex]: TAlphaColor;
grid.FontStyles[ColumnIndex, RowIndex]: TFontStyle;
grid.FontNames[ColumnIndex, RowIndex]: string;
grid.FontSizes[ColumnIndex, RowIndex]: integer;
```

The above information applies to default cells (in the grid of the class TTMSFMXGridCell) but you can actually add any type of FireMonkey TFMXObject as a cell in the grid.This can be done in following way with the event OnGetCellClass:

```
procedure TForm1.TMSFMXGrid1GetCellClass(Sender: TObject; ACol, ARow:
Integer;
  var CellClassType: TFmxObjectClass);
begin
  if (ACol = 3) and (ARow >= TMSFMXGrid1.FixedRows) then
    CellClassType := TTMSFMXCheckGridCell;

  if (ACol = 4) and (ARow >= TMSFMXGrid1.FixedRows) then
    CellClassType := TButton;
end;
```

This code snippet specifies a checkbox (a specific grid adapted checkbox that can have a background color) for column 3 and a button for column 4. Note that using OnGetCellClass means that the grid itself will be responsible for the creation of the cell control. It is equally possible to insert any type of control in a cell that was created outside the grid via the event OnGetCellControl.
This allows for an unprecedented flexibility to customize the grid. For example, inserting a grid in a grid cell is equally simple this way:

```
procedure TForm1.TMSFMXGrid1GetCellControl(Sender: TObject; ACol,
ARow: Integer;
  var AControl: TFmxObject);
begin
  if (ACol = 2) and (ARow = 2) then
    AControl := MySubGrid;
end;
```

Several methods are provided as well that allow to add controls to cells programmatically:

```
grid.AddCheckBox()
grid.AddDataCheckBox()
grid.AddRadioButton()
grid.AddBitmap()
grid.AddNode()
grid.CellControls[ColumnIndex, RowIndex]: TControl
```

## Cell merging

| 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 1 | Alfa Romeo | 156 1,6TS | 1598 | 88 |
| 2 | Alfa Romeo | 156 1,8TS | 1774 | 106 |
| 3 | Alfa Romeo | 156 2,0TS | 1970 | 114 |
| 4 | Alfa Romeo | 156 2,5 | 2492 | 140 |
| 5 | Alfa Romeo | 166 2,0TS | 1970 | 114 |
| 6 | Alfa Romeo | 166 2,0V6 | 1996 | 151 |
| 7 | Alfa Romeo | 166 2,5V6 | | |
| 8 | Alfa Romeo | | | |
| 9 | Alfa Romeo | Spider 1,8 | 1747 | 106 |
| 10 | Alfa Romeo | Spider 2,0 | 1970 | 114 |
| 11 | Alfa Romeo | Spider 3,0 | 2959 | 141 |

*Use of merged cells in the grid*

The grid has built-in support for cell merging. Cell merging means that several adjacent cells are merged together to form a new single cell. The access to a merged cell is done via the top left cell. Cell merging and the reverse action, cell splitting is easy:

```
  grid.MergeCells(Col, Row, ColumnCount, RowCount: integer);
  grid.SplitCell(Col,Row: integer);
```

and several helper functions exist:

```
  grid.IsMergedCell(), grid.RowSpan(), grid.ColSpan(),
grid.BaseCell().
```

## Editing the grid

The grid supports different inplace editors as well as the capability to choose any type of FireMonkey control as inplace cell edit control. By default, a cell is edited with a regular TEdit control. Different editor types can be selected from the built-in edit controls via the event OnGetCellEditorType:

```
procedure TForm1.TMSFMXGrid1GetCellEditorType(Sender: TObject; ACol,
  ARow: Integer; var CellEditorType: TTMSFMXGridEditorType);
begin
  case acol of
  1: CellEditorType := etNumericEdit;
  2: CellEditorType := etNumericEditBtn;
  3: CellEditorType := etComboBox;
  4: CellEditorType := etDatePicker;
```

```
  5: CellEditorType := etCustom;
  end;
end;
```

The TTMSFMXGridEditorType defines many types of already built-in editor types in the grid. This includes different kind of editors for string, numeric, float, hex type, a spin editor, date picker, color picker, combobox, trackbar and dial. When using the OnGetCellEditorType, normally nothing else is required to handle editing. If in addition validation is needed, the event OnCellEditValidateData can be implemented. This event is triggered when editing is about to stop. It passes the new edited value together with an Allow parameter. In this example event handler, the validation is performed that the maximum length cannot exceed 10:

```
procedure TForm1.TMSFMXGrid1CellEditValidateData(Sender: TObject;
ACol,
  ARow: Integer; CellEditor: TFmxObject; var CellString: string;
  var Allow: Boolean);
begin
  Allow := Length(CellString) <= 10;
end;
```

If another type of FireMonkey control is needed as inplace editor that is not listed in TTMSFMXGridEditorType, the CellEditorType can be set to etCustom. In this case, the grid will trigger the event OnGetCellEditorCustomClassType. In this event, it can be specified via the parameter CellEditorCustomClassType what the class is of the inplace editor:

```
procedure TForm1.TMSFMXGrid1GetCellEditorCustomClassType(Sender:
TObject; ACol,
  ARow: Integer; var CellEditorCustomClassType: TFmxObjectClass);
begin
  if ACol = 5 then
    CellEditorCustomClassType := TMyVerySpecialCustomEditor;
end;
```

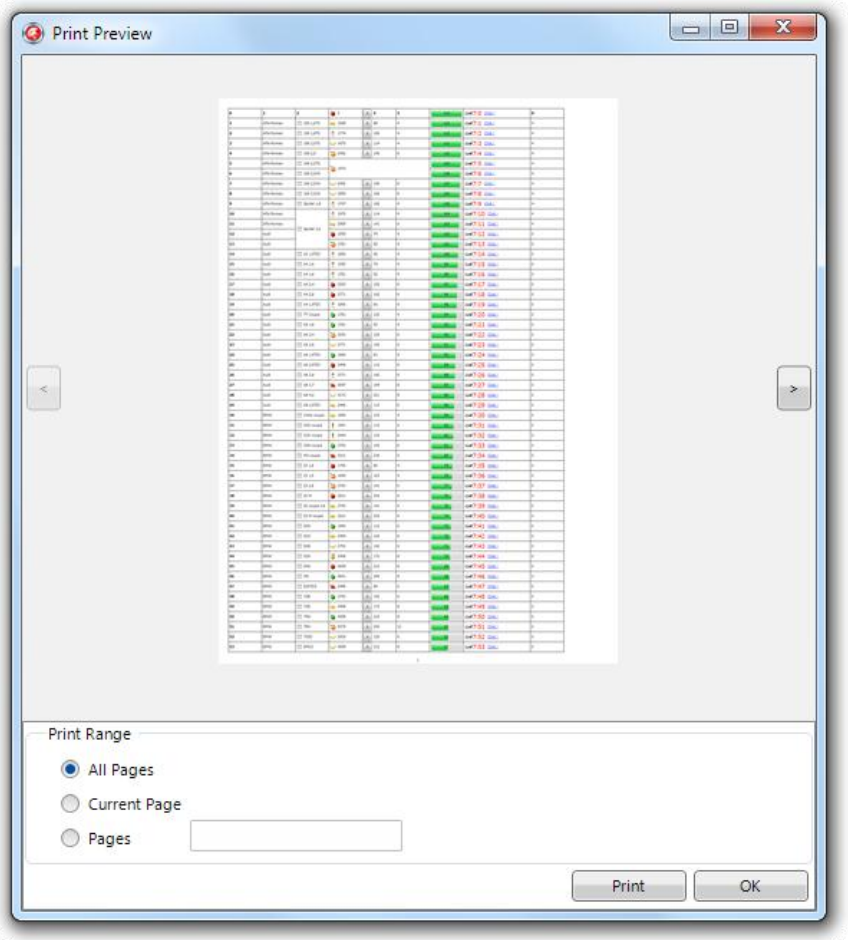
To map the cell content to the custom inplace editor and vice versa, the events OnCellEditGetData, OnCellEditSetData can be used.

```
procedure TForm1.TMSFMXGrid1CellEditGetData(Sender: TObject; ACol,
  ARow: Integer; CellEditor: TFmxObject; var CellString: string);
begin
  if (ACol = 5) then
  begin
    (CellEditor as TMyVerySpecialCustomEditor).Data := CellString;
  end;
end;

procedure TForm1.TMSFMXGrid1CellEditSetData(Sender: TObject; ACol,
  ARow: Integer; CellEditor: TFmxObject; var CellString: string);
begin
  if (ACol = 5) then
```

```
  begin
    CellString := (CellEditor as TMyVerySpecialCustomEditor).Data;
  end;
end;
```

## Printing, print preview, print to image



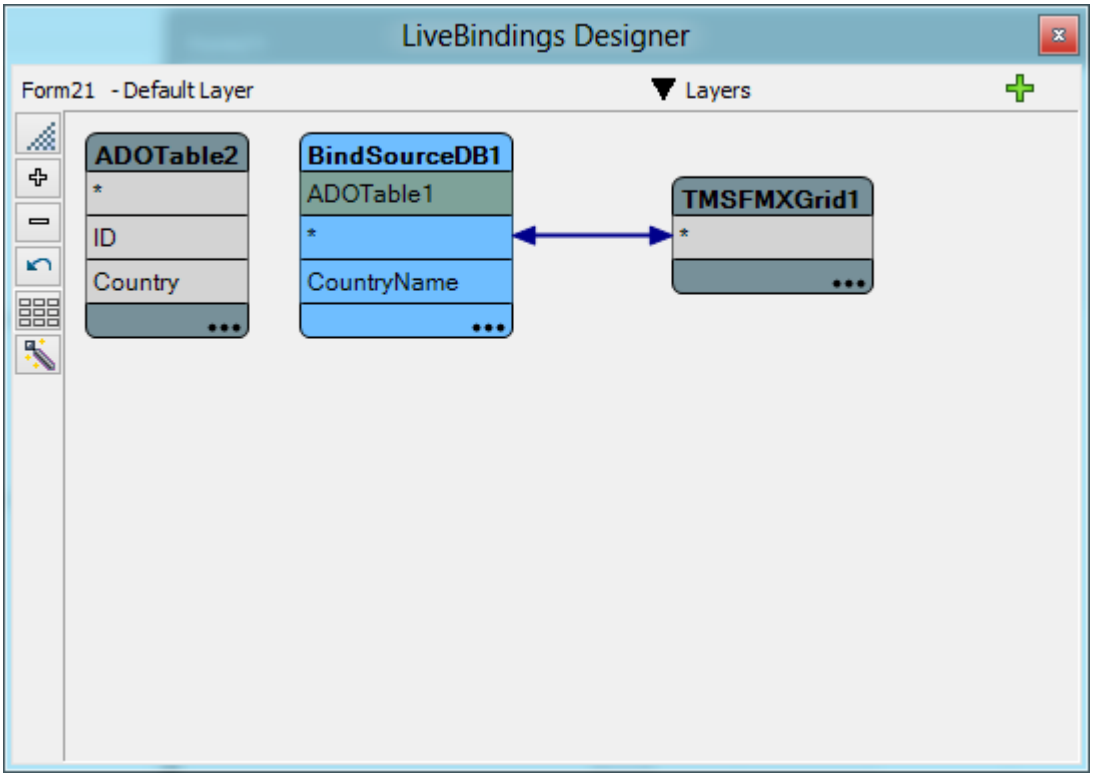

*The included print preview dialog for the grid*

Even when we're living in the digital age, for a lot of people, data only exists when it is on paper. The grid being the primary tool of data presentation and organization features as such also a printing capability. The full grid or a range of cells can be printed. Printing is as easy as calling grid.Print but many more equivalent methods are available like: grid.PrintPageSelection(), grid.PrintPageFromTo() and many more to limit nr. of cells to print, limit nr. of pages etc.. As a helper, there is also a print preview dialog component: TTMSFMXPrintPreviewDialog that is simple in its use. Assign a grid to the TMSFMXPrintPreviewDialog.Grid property and call its Execute method.
Finally, the print output can also be rendered to an image file or a memory bitmap canvas. The

method grid.PrintPageToImage(FileName:string) can generate .BMP, .PNG, .GIF and .JPG files. The file format simply depends on the extension of the filename specified.

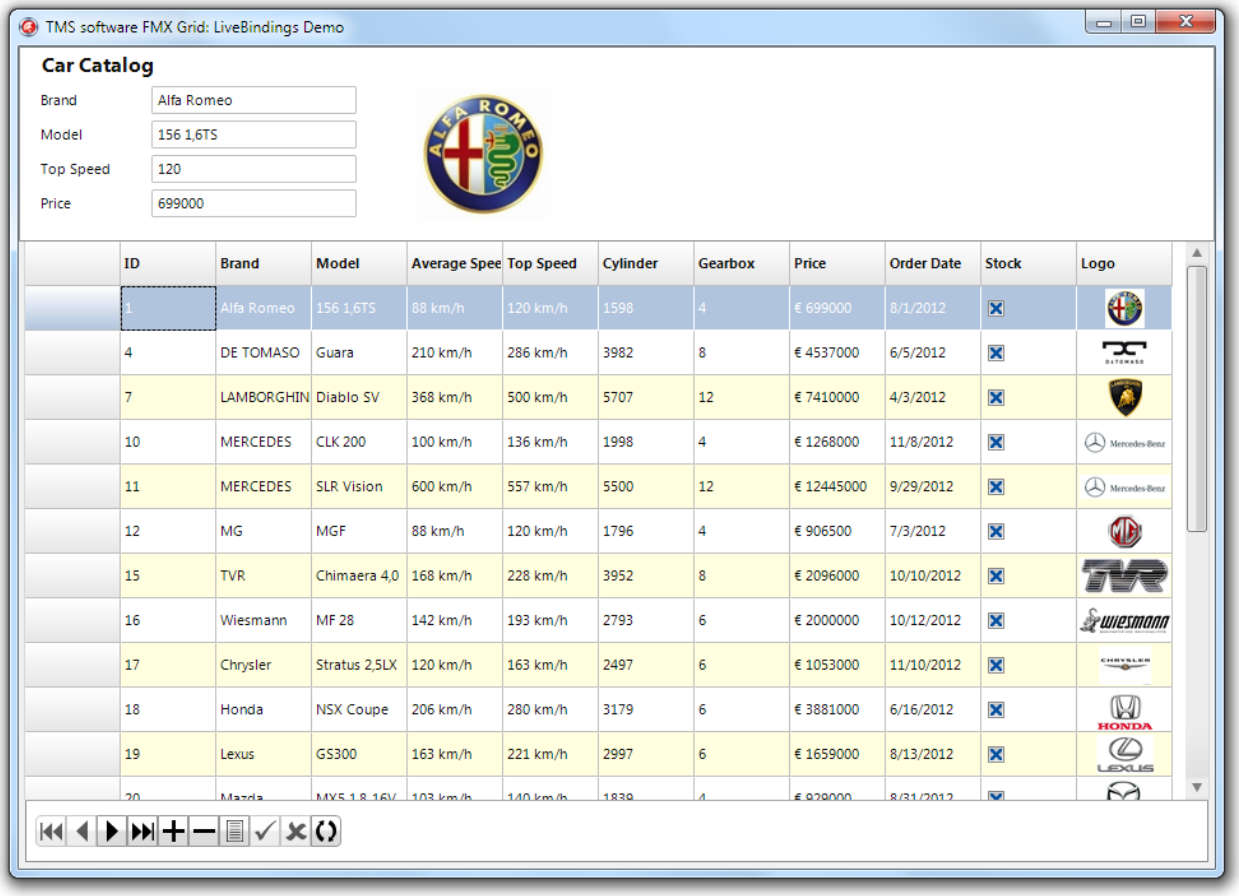
## LiveBindings

The grid fully supports LiveBindings to display, navigate in and edit datasets. In XE3, the most easy way to do this is via Visual LiveBindings. To get started, drop a TTMSFMXGrid on the form and a dataset. Right-click on the form and select “Bind Visually” to start the Visual LiveBindings editor. Either connect the fields you want to have in the grid from the dataset to the grid or connect the ‘*’ item from the dataset to the ‘*’ item from the grid. This way, the grid will display all fields.




*Visual LiveBindings editor in the IDE*

The grid will automatically display Boolean fields as checkboxes, graphic or blob fields as images and memo fields as memo text in cells when possible. It will also by default select a numeric edit control as inplace editor for numeric fields, a datepicker for date fields or a memo for memo fields. You can override this at any time though by implementing the OnGetEditorType event.




*Grid connected to a dataset: checkboxes for boolean fields, displays images from BLOBs*

This is Visual LiveBindings with the TMS grid in its most easy form. Of course you can also use the grid in more complex binding expression setup scenarios. In the full grid documentation, it is covered for example how you can setup the grid and LiveBindings to have a combobox lookup inplace editor. It is for example also possible to automatically insert or remove records in the dataset from a bound grid when grid.Options.Keyboard.DeleteKeyHandling = dkhDeleteRow or grid.Options.Keyboard.InsertKeyHandling = dkhInsertRowAfter.

## Conclusion

In this article, we have tried to give a background on the requirements and decisions that were made in the creation of the TMS Grid for FireMonkey and give an overview of its capabilities. Each topic covered in this overview article can be elaborated in much more detail and depth unveiling the power of the grid. Please see the TMS Grid for FireMonkey developers guide for this.

## Your advantages to upgrade to TMS Pack for FireMonkey

- **Full source code with support for Delphi XE2, XE3, C++Builder XE2, XE3 & Windows, Mac OS-X, iOS platforms**
- **All additional components available in the TMS Pack for FireMonkey**
- **Free updates of TMS Pack for FireMonkey from current version v1.6 till v2.6**
- **Access to new components our team has in development for future updates**
- **Use of our priority support services during a full version cycle**
- **Further extensions, new features & enhancements to existing components**
- **Updates for upcoming new Delphi & C++Builder versions**

**For more information, visit:** http://www.tmssoftware.com/site/tmsfmxpack.asp

## Discount offers valid until December 31, 2012.

-30%

**TMS Pack for FireMonkey**
Single developer license

Discount online order form

-40%

**TMS Pack for FireMonkey**
Site license

Discount online order form