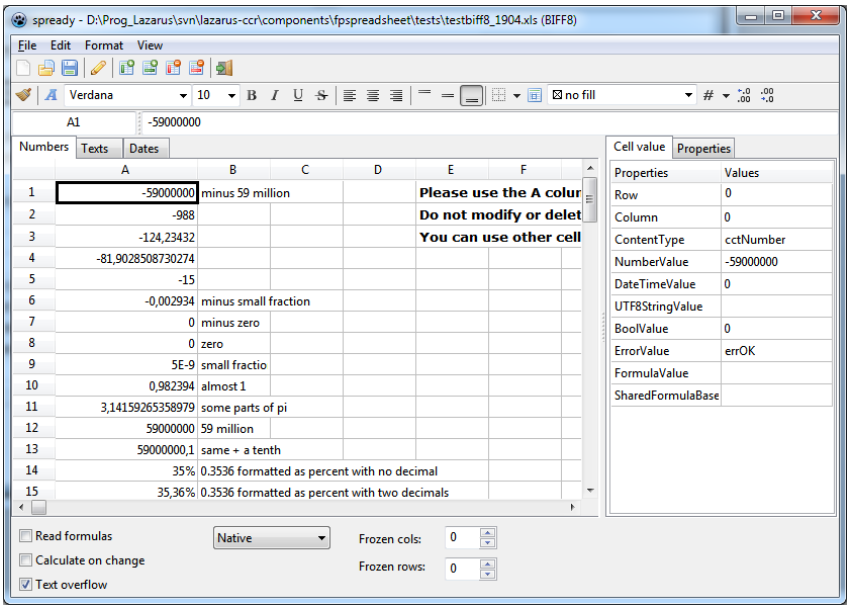
FPSpreadsheet

From Lazarus wiki

| Deutsch (de) | English (en) | español (es) | français (fr) | polski (pl) | русский (ru) |

The fpSpreadsheet library offers a convenient way to generate and read spreadsheet documents in various formats. The library is written in a very flexible manner, capable of being extended to support any number of formats easily.

Screenshot of spready demo program provided with fpspreadsheet showing an XLS file:



Contents

Documentation

This wiki page covers the latest development/trunk version of FPSpreadsheet available via subversion. Please see section Stable releases for documentation on the stable version that you can download.

API Documentation

API Reference

A help file in CHM format can be found in the FPSpreadsheet installation folder *docs*. If you did not yet install the package follow http://lazarus-ccr.svn.sourceforge.net/viewvc/lazarus-ccr/components/fpspreadsheet/docs/fpspreadsheet-api.chm (http://lazarus-ccr.svn.sourceforge.net/viewvc/lazarus-ccr/components/fpspreadsheet/docs/fpspreadsheet-api.chm) to *fpspreadsheet-api.chm*.

The second CHM file available in the folder *docs*, *fpspreadsheet-wiki.chm*, is a snapshot of the FPSpreadsheet-related wiki pages bundled into a single help file.

Basics

The smallest entities in a spreadsheet are the **cells** which contain the data. Cells can hold various data types, like strings, numbers, dates, times, boolean values, or formulas. In addition, cells can contain information on formatting, i.e. font style, background color, text alignment etc.

The cells are arranged in a grid-like structure, called **worksheet**, or **spreadsheet**, consisting of **rows** and **columns**. Each cell has a unique address given by the row and column index.

Worksheets are bound together to form a **workbook** which represents the document of the spreadsheet application. The workbook also stores information that is needed from all worksheets, i.e. font list, cell and number format lists, etc.

FPSpreadsheet follows this same structure - there is a `TCell`, a `TsWorksheet`, and a `TsWorkbook`.

Workbook

The class `TsWorkbook` is the main class visible to the user. It provides methods for reading data from and writing to files. The versatile structure of the library provides access to various popular file formats, like Excel *.xls* or *.xlsx*, or OpenOffice/LibreOffice *.ods*.

The file format is specified by the type `TsSpreadsheetFormat` defined in the unit `fpstypes`

```
type
  TsSpreadsheetFormat = (sfExcel2, sfExcel5, sfExcel8, sfExcelXML, sfOOXML,
    sfOpenDocument, sfCSV, sfHTML, sfWikiTable_Pipes, sfWikiTable_WikiMedia, sfUser);
```

where

- `sfExcel2, sfExcel5, sfExcel8` stands for versions of the binary xls format used by Excel ("BIFF" = "Binary Interchange File Format") with `sfExcel8` being the most modern one.
- `sfOOXML` corresponds to the newer xlsx format introduced by Excel2007
- `sfExcelXML` is the xml format which was introduced by Microsoft for Office XP and 2003. Not very popular.
- `sfOpenDocument` is the spreadsheet format used by OpenOffice/LibreOffice; by default, the files have the extension *.ods*.
- `sfCSV` refers to comma-delimited text files (default extension *.csv*); they can be understood by any text editor and all spreadsheet programs, but do not contain formatting information.
- `sfHTML` denotes the standard HTML format as used in web browsers.
- `sfWikiTable_Pipes` and `sfWikiTable_WikiMedia` is the format used by tables in wiki websites.
- `sfUser` is needed to register a user-defined format. There are no plans to implement "ancient" file formats like Excel3.0/4.0 or Lotus. It is possible, however, to provide your own reading and writing classes to extend the functionality of FPSpreadsheet - see the section below on Adding new file formats

When applying fpspreadsheet the first task is to create an instance of the workbook:

```
var
  MyWorkbook: TsWorkbook;
begin
  MyWorkbook := TsWorkbook.Create;
  ...
```

Reading of spreadsheet files is accomplished (among others) by the workbook methods

- `procedure ReadFromFile(AFileName: string)`:
  Reads the file with the given name and automatically determines the correct file format.
- `procedure ReadFromFile(AFileName: string; AFormat: TsSpreadsheetFormat)`:
  Reads the file, but assumes that the file format is as specified by `AFormat`.

The following workbook methods can be used for saving to file:

- `procedure WriteToFile(const AFileName: string; const AFormat: TsSpreadsheetFormat; const AOverwriteExisting: Boolean = False)`:
  Writes the workbook to the given file using the given spreadsheet format. If the file already exists it is automatically overwritten if `AOverwriteExisting` is `true`:
- `procedure WriteToFile(const AFileName: String; const AOverwriteExisting: Boolean = False)`:
  ditto, but the file format is determined from the file extension provided (in case of Excel's xls the most recent version, `sfExcel8`, is used).

After calling these methods it is advantageous to look at the workbook's property `ErrorMsg` in which messages due to errors or warnings are collected that might have occured during reading/writing. This property returns a multi-lined string which is displayed best in a memo component; if everything was fine it is an empty string.

Note: FPSpreadsheets provides specialized units for reading from and writing to each file format. These units are not available automatically, you have to add them to the uses clause explicitly. FPSpreadsheet will complain about "unsupported file format" if the requested reader/writer is not found. Here is a list of the unit names:

- *xlsbiff2*, *xlsbiff5* and *xlsbiff8* for the binary xls file formats `sfExcel2`, `sfExcel5` and `sfExcel8`, respectively,
- *xlsxOOXML* for the xlsx file format `sfOOXML` of Excel 2007 and later,
- *xlsXML* for the xml format of Excel XP and 2003,
- *fpsopendocument* for the file format `sfOpenDocument` of OpenOffice/LibreOffice,
- *fpsCSV* for text files with comma-separated values (csv),
- *fpsHTML* for HTML files,
- *wikitables* for `sfWikiTable_Pipes` and `sfWikiTable_WikiMedia`,
- or, simply add *fpsallformats* to get read/write support for all file formats supported.

Worksheet

The workbook contains a list of `TsWorksheet` instances. They correspond to the tabs that you see in Excel or Open/LibreOffice. When reading a spreadsheet file the worksheets are created automatically according to the file contents. When a spreadsheet is created manually to be stored on file a worksheet has to be created by *adding* it to the workbook:

```
var
  MyWorkbook: TsWorkbook;
  MyWorksheet: TsWorksheet;
begin
  MyWorkbook := TsWorkbook.Create;
  MyWorksheet := MyWorkbook.AddWorksheet('My_Table');
    // 'My_Table' is the "name" of the worksheet
  ...
```

Already existing worksheets can be accessed by using the `TsWorkbook` methods

- `function GetFirstWorksheet: TsWorksheet`: retrieves the first worksheet of the workbook.
- `function GetWorksheetByIndex(AIndex: Cardinal): TsWorksheet`: returns the worksheet with the given index (starting at 0).
- `function GetWorksheetByName(AName: String): TsWorksheet`: returns the worksheet with the given name which was used when the worksheet was added.

The count of already existing worksheets can be queried by calling `GetWorksheetCount`.

Cell

The worksheet, finally, gives access to the cells. A newly created worksheet, as in above example, is empty and does not contain any cells. Cells are added by assigning data or attributes to them by one of the `WriteXXXX` methods of the worksheet. As already mentioned, a cell is addressed by the index of the row and column to which it belongs. As usual, row and column indexes start at 0. Therefore, cell "A1" belongs to row 0 and column 0. It should be noted that row and column index are always specified in this order, this is *different from the convention of* `TStringGrid`. The following example creates a cell at address A1 and puts the number `1.0` in it.

```
var
  MyWorkbook: TsWorkbook;
  MyWorksheet: TsWorksheet;
begin
  MyWorkbook := TsWorkbook.Create;
  MyWorksheet := MyWorkbook.AddWorksheet('My_Table');
  MyWorksheet.WriteNumber(0, 0, 1.0);  // "A1" has row=0 and col=0
  ...
```

It is also possible to access cells directly by means of the methods `FindCell(ARow, ACol)` or `GetCell(ARow, ACol)` of the workbook. Both funtions exist also in an overloaded version to which the cell address can be passed in Excel notation: `FindCell(ACellStr: String)` or `GetCell(ACellStr: String)`. Please be aware that these functions return a `pointer` to the cell data (type `PCell`). Don't forget to dereference the pointers! The difference between `FindCell` and `GetCell` is that the former one returns `nil`, if a cell does not yet exist, while the latter one creates an empty cell in this case.

```
if MyWorksheet.FindCell('A1') = nil then
```

```
  WriteLn('Cell A1 does not exist.');
```

The basic TCell record

This is the declaration of the cell's data type:

```
type
  TCell = record
    { Location of the cell }
    Worksheet: TsWorksheet;
    Col: Cardinal; // zero-based
    Row: Cardinal; // zero-based

    { Index of format record }
    FormatIndex: Integer;

    { Status flags }
    Flags: TsCellFlags;        // (cfHasComment, cfMerged, cfHyperlink, ...)

    { Cell content }
    UTF8StringValue: String;   // strings cannot be part of a variant record
    case ContentType: TCellContentType of  // must be at the end of the declaration
      cctEmpty      : ();      // has no data at all
      cctFormula    : ();      // UTF8StringValue is outside the variant record
      cctNumber     : (Numbervalue: Double);
      cctUTF8String : ();      // FormulaValue is outside the variant record
      cctDateTime   : (DateTimevalue: TDateTime);
      cctBool       : (BoolValue: boolean);
      cctError      : (ErrorValue: TsErrorValue);
  end;
  PCell = ^TCell;
```

The field `ContentType` indicates which data type is stored in the cell:

```
type
  TCellContentType = (cctEmpty, cctFormula, cctNumber, cctUTF8String, cctDateTime, cctBool, cctError);
```

According to this field the corresponding data can be found in the fields

- `NumberValue` (for `ContentType=cctNumber`), or
- `UTF8StringValue` (for `ContentType=cctUTF8String`), or
- `DateTimeValue` (for `ContentType=cctDateTime`), or
- `BoolValue` (for `ContentType=cctBool`), i.e. `TRUE` or `FALSE`, or
- `ErrorValue` (for `ContentType=cctError`).

Due to usage of a variant record most of these values are overlapping, i.e. modification of `NumberValue` affects also the other values. Therefore, always respect the `ContentType` when accessing the `TCell` record directly (the worksheet methods discussed below consider this automatically).

The field `Flags` tells whether additional data are associated with the cell which are not included in the cell record usually to save memory:

```
type
  TsCellFlag = (cfHasComment, cfHyperlink, cfMerged, cfHasFormula, cf3dFormula);
  TsCellFlags = set of TsCellFlag;
```

- `cfHasComment`: A comment record can be found in the `Comments` of the worksheet.
- `cfHyperlink`: The cell is contains a hyperlink stored in the `Hyperlinks` of the worksheet.
- `cfMerged`: The cell belongs to a merged block and extends across several cells.
- `cfHasFormula`: The cell is associated with a formula which is stored in the `Formulas` of the worksheet.
- `cf3dFormula`: The formula associated with the cell contains elements referencing other sheets of the same workbook.

Note: After calculation of a formula or after reading of a file, the `ContentType` of the formula cell is converted to that of the result. Then the presence of a formula can only be detetected by calling the function `HasFormula(cell)` for the cell to be queried (`cell: PCell`) in unit *fpsUtils*; this function checks the presence of the element `cfHasFormula` in the cell flags.

How to add and read data to/from a cell

Adding values to a cell is most easily accompished by using one of the `WriteXXXX` methods of the worksheet. The most important ones are:

```
type
  TsWorksheet = class
    ...
    { Writing of currency values }
    function WriteCurrency(ARow, ACol: Cardinal; AValue: Double;
      ANumFormat: TsNumberFormat = nfCurrency; ADecimals: Integer = 2;
      ACurrencySymbol: String = '?'; APosCurrFormat: Integer = -1;
      ANegCurrFormat: Integer = -1): PCell; overload;
    procedure WriteCurrency(ACell: PCell; AValue: Double;
      ANumFormat: TsNumberFormat = nfCurrency; ADecimals: Integer = -1;
      ACurrencySymbol: String = '?'; APosCurrFormat: Integer = -1;
      ANegCurrFormat: Integer = -1); overload;
    function WriteCurrency(ARow, ACol: Cardinal; AValue: Double;
      ANumFormat: TsNumberFormat; ANumFormatString: String): PCell; overload;
    procedure WriteCurrency(ACell: PCell; AValue: Double;
      ANumFormat: TsNumberFormat; ANumFormatString: String); overload;

    { Writing of date/time values }
    function WriteDateTime(ARow, ACol: Cardinal; AValue: TDateTime): PCell; overload;
    procedure WriteDateTime(ACell: PCell; AValue: TDateTime); overload;
    function WriteDateTime(ARow, ACol: Cardinal; AValue: TDateTime;
      ANumFormat: TsNumberFormat; ANumFormatStr: String = ''): PCell; overload;
    procedure WriteDateTime(ACell: PCell; AValue: TDateTime;
      ANumFormat: TsNumberFormat; ANumFormatStr: String = ''); overload;
    function WriteDateTime(ARow, ACol: Cardinal; AValue: TDateTime;
      ANumFormatStr: String): PCell; overload;
    procedure WriteDateTime(ACell: PCell; AValue: TDateTime;
      ANumFormatStr: String); overload;

    { Writing of number values }
    function WriteNumber(ARow, ACol: Cardinal; ANumber: double): PCell; overload;
    procedure WriteNumber(ACell: PCell; ANumber: Double); overload;
    function WriteNumber(ARow, ACol: Cardinal; ANumber: double;
      ANumFormat: TsNumberFormat; ADecimals: Byte = 2): PCell; overload;
    procedure WriteNumber(ACell: PCell; ANumber: Double;
      ANumFormat: TsNumberFormat; ADecimals: Byte = 2); overload;
    function WriteNumber(ARow, ACol: Cardinal; ANumber: double;
      ANumFormat: TsNumberFormat; ANumFormatString: String): PCell; overload;
    procedure WriteNumber(ACell: PCell; ANumber: Double;
      ANumFormat: TsNumberFormat; ANumFormatString: String); overload;

    { Writing of string values }
    function WriteText(ARow, ACol: Cardinal; AText: ansistring;
      ARichTextParams: TsRichTextParams = nil): PCell; overload;
    procedure WriteText(ACell: PCell; AText: String;
      ARichTextparams: TsRichTextParams = nil); overload;

    // the old string methods "WriteUTF8Text" are deprecated now
    ...
```

Some of these methods exist in overloaded versions in which cell formatting parameters can be added together with the cell value. Correspondingly to writing, there is also a number of `worksheet` methods for reading the cell values:

```
type
  TsWorksheet = class
    ...
    { Reading cell content as a string }
    function  ReadAsText(ARow, ACol: Cardinal): string; overload;
    function  ReadAsText(ACell: PCell): string; overload;
    function  ReadAsText(ACell: PCell; AFormatSettings: TFormatSettings): string; overload;

    { Reading cell content as a number }
    function  ReadAsNumber(ARow, ACol: Cardinal): Double; overload;
    function  ReadAsNumber(ACell: PCell): Double; overload;
    function  ReadNumericValue(ACell: PCell; out AValue: Double): Boolean;

    { Reading cell content as a date/time value }
    function  ReadAsDateTime(ARow, ACol: Cardinal; out AResult: TDateTime): Boolean; overload;
    function  ReadAsDateTime(ACell: PCell; out AResult: TDateTime): Boolean; overload;
    ...
```

Index to Cell format

`FormatIndex` is the index of the cell format record. It describes the formatting attributes of a cell. These records are collected by an internal list of the workbook and are defined like this:

```
type
  TsCellFormat = record
    FontIndex: Integer;
    TextRotation: TsTextRotation;
    HorAlignment: TsHorAlignment;
    VertAlignment: TsVertAlignment;
    Border: TsCellBorders;
    BorderStyles: TsCellBorderStyles;
    Background: TsFillPattern;
    NumberFormatIndex: Integer;
    NumberFormat: TsNumberFormat;
    NumberFormatStr: String;
    BiDiMode: TsBiDiMode;          // bdDefault, bdLTR {left-to-right}, bdRTL {right-to-left)
    Protection: TsCellProtection;   // cpLockCell, cpHideFormulas
    UsedFormattingFields: TsUsedFormattingFields;
      //uffTextRotation, uffFont, uffBold, uffBorder, uffBackground, uffNumberFormat, uffWordWrap, uffHorAlign, uffVertAlign, uffBiDi, uffProtection, uffDoNotPrint
  end;
```

- `FontIndex`: text font by specifying the index in the workbook's font list
- `TextRotation`: specifies whether the cell text is written horizontally or vertically
- `HorAlignment`: left-aligned, horizontally centered, or right-aligned text
- `VertAlignment`: top, bottom or vertically centered text
- `Border`: a set of flags indicating that - if set - a border line is drawn at the left, top, right, or bottom cell edge. The lines are drawn according to the `BorderStyles` which define the linestyle and color of the border.
- `Background`: a record defining the background fill of a cell (pattern style, pattern color, and background color - see chapter on cell background below).
- `NumberFormat` and `NumberFormatStr` specify how number or date/time values are formatted (e.g., number of decimal places, long or short date format, etc.).
- Only those format attributes for which a flag is set in the `UsedFormattingFields` are considered when formatting a cell. If a flag is not included then the corresponding attribute is ignored and replaced by its default value.

For specifying a format for a given cell call the corresponding the worksheet method `WriteXXXX`, for retrieving a format call `ReadXXXX`. These methods usually get a pointer to the cell as a parameter, but there are also overloaded versions which accept the row and column index. Moreover, formatting styles can also be applied directly to the cell by using a record helper implemented in unit *fpsCell*.

See cell formatting below for a more detailed description.

Columns and rows

Column and row records are added for each column and row having a non-default properties:

```
type
  TCol = record
    Col: Cardinal;
    Width: Single;
    ColWidthType: TsColWidthType;     // = (cwtDefault, cwtCustom)
    FormatIndex: Integer;
    Options: TsColRowOptions;         // set of [croHidden, croPageBreak]
  end;
  PCol = ^TCol;

  TRow = record
    Row: Cardinal;
    Height: Single;
    RowHeightType: TsRowHeightType;   // = (rhtDefault, rhtCustom, rhtAuto)
    FormatIndex: Integer;
    Options: TsColRowOptions;         // set of [croHidden, croPageBreak]
    Hidden: Boolean;
    PageBreak: Boolean;
  end;
  PRow = ^TRow;
```

Column width / row height

Column widths and **row heights** can be specified in a variety of units defined by the type `TsSizeUnits = (suChars, suLines, suMillimeters, suCentimeters, suPoints, suInches)`. `suChars` refers to the count of `0` characters fitting into the column width - this is the way how Excel defines column widths. `suLines` is the number of lines fitting into the row height. Both units are based on the character size of the workbook's default font. The other units are conventional physical length units (1 cm = 10 mm, 1 inch = 25.4 mm = 72 pt). Fractional values are accepted. The workbook and worksheets store lengths internally in millimeters (`MyWorkbook.Units`).

The Office applications usually adjust the row heights automatically according to the font or text rotation of the cell content. This case is identified by `RowHeightType` having the value `rhtAuto`. Since the worksheet cannot calculate text size very accurately automatic row heights are not written by FPSpreadsheet; they are replaced by the **default row height**. The default row height is also used if a row is empty, i.e. does not contain any data cells. Its value can be changed by calling the worksheet's `WriteDefaultRowHeight()` or by using the worksheet property `DefaultRowHeight`. In `WriteDefaultRowHeight`, the units must be specified while in `DefaultRowHeight` they are assumed to be lines. Similarly, the **default column width** can be specified by `WriteDefaultColWidth()` or the property `DefaultColWidth` (in characters).

In order to overrun automatic and default row heights call the worksheet method `WriteRowHeight()`. These row records are identified by `RowHeightType` having the value `rhtCustom`. In the same way the width of columns can be set to a specific value by calling `WriteColWidth()`. `ColWidthType` of these columns is `cwtCustom`.

The height/width of a particular row/column can be retrieved by means of the methods `GetRowHeight` or `GetColWidth`. Note that these methods return the default row heights/column widths if there are no `TRow`/`TCol` records.

```
type TsWorksheet = class
  ...
  { Set row height }
  procedure WriteRowHeight(ARowIndex: Cardinal; AHeight: Single; AUnits: TsSizeUnits);
  { Set column width }
  procedure WriteColWidth(AColIndex: Cardinal; AWidth: Single; AUnits: TsSizeUnits);
  { Set default row height }
  procedure WriteDefaultRowHeight(AHeight: Single; AUnits: TsSizeUnits);
```

```
  {Set default column width }
  procedure WriteDefaultColWidth(AWidth: Single; AUnits: TsSizeUnits);

  { Return row height }
  function GetRowHeight(ARowIndex: Cardinal; AUnits: TsSizeUnits): Single;
  { Return column width }
  function GetColWidth(AColIndex: Cardinal; AUnits: TsSizeUnits): Single;
  { Return default row height }
  function ReadDefaultRowHeight(AUnits: TsSizeUnits): Single;
  { Return default column width }
  function ReadDefaultColWidth(AUnits: TsSizeUnits): Single;

  property DefaultRowHeight: Single;  // in lines
  property DefaultColWidht: Single;   // in characters
```

There are also overloaded versions of these methods which do not require the `AUnits` parameter. In this case, row heights are defined in terms of line count, and column widths are defined in terms of character count. Note that these variants are from previos versions and are deprecated now.

Column and row formats

The `FormatIndex` element of the row and column records format applied to the entire row or column. Together with cells, these formats are stored as `TsCellFormat` records in an internal workbook list, `FormatIndex` is the index of the format properties into this list. See details in section Cell formatting. **Row and column formats** are primarily applied to empty cells, but if a new cell is added it will automatically get the format of the row or column. (If both row and column have different formats then the row format will be used).

Here is a listing of the worksheet methods available for column and row formatting:

```
type TsWorksheet = class
  // Assign a format to a column or row
  procedure WriteColFormatIndex(ACol: Cardinal; AFormatIndex: Integer);
  procedure WriteRowFormatIndex(ARow: Cardinal; AFormatIndex: Integer);

  // Query the format of a column or row
  function  GetColFormatIndex(ACol: Cardinal): Integer;
  function  GetRowFormatIndex(ARow: Cardinal): Integer;

  // Returns the font assigned to a column or row
  function  ReadColFont(ACol: PCol): TsFont;
  function  ReadRowFont(ARow: PRow): TsFont;

  // Check whether any column or row uses a special format
  function  HasRowFormats: Boolean;
  function  HasColFormats: Boolean;
```

Hidden columns / rows

Adding the flag `croHidden` to the row's or column's `Options` hides the row or column in the Office application (or in TsWorksheetGrid). The following worksheet methods are helpers to handle **row/column visibility**:

```
type TsWorksheet = class
  ...
  { Hide column/row }
  procedure HideCol(ACol: Cardinal);
  procedure HideRow(ARow: Cardinal);

  { Show a previously hidden column/row }
  procedure ShowCol(ACol: Cardinal);
  procedure ShowRow(ARow: Cardinal);

  { Check whether column/row is hidden }
  function  ColHidden(ACol: Cardinal): Boolean;
  function  RowHidden(ARow: Cardinal): Boolean;
```

Page breaks

The flag `croPageBreak` can be added to the row's or column's `Options` in order to force a **page break** before the corresponding row or column when the worksheet is printed by the Office applications. Note that FPSpreadsheet itself does not support printing. The following worksheet methods help with page breaks:

```
type TsWorksheet = class
  ...
  { Enforce a page break before the specified column/row }
  procedure AddPageBreakToCol(ACol: Cardinal);
  procedure AddPageBreakToRow(ARow: Cardinal);

  { Removes a page break previously added to a column/row }
  procedure RemovePageBreakFromCol(ACol: Cardinal);
  procedure RemovePageBreakFromRow(ARow: Cardinal);

  { Checks whether a page break is forced before the specified column/row }
  function IsPageBreakCol(ACol: Cardinal): Boolean;
  function IsPageBreakRow(ARow: Cardinal): Boolean;
```

Events

Worksheets and workbooks fire a series of events such as OnChangeCell, OnChangeFont, etc. The events usually are intended for interaction with the visual spreadsheet controls. If you need to attach your own handlers you should make sure that the original handler is called, at least in a gui program using the spreadsheet controls. Or you use the events provided by the visual controls themselves, e.g. `WorksheetGrid.OnEditingDone` instead of `WorksheetGrid.Worksheet.OnChangeCell`.

Formulas

Two kinds of formulas are supported by FPSpreadsheet:

- **String formulas**: These are written in strings just like in the office applications, for example `"=ROUND(A1+B1,0)"`. They are used internally in the files of Open/LibreOffice and Excel *.xlsx*.
- **RPN formulas** are used internally by the binary *.xls* Excel files. They are written in Reverse Polish Notation (RPN), for example: `A1, B1, Add, 0, ROUND`. If a spreadsheet containing formulas is to be saved in a binary Excel format, the RPN formulas required are generated automatically.

FPSpreadsheet can convert between string and rpn formulas. Formulas in both types can be calculated.

In older versions of FPSpreadsheet, formulas were stored directly in the cell record. This was given up to have parsed formulas available for faster calculation. The formulas are stored in the tree `Formulas` of the worksheet. The formula record contains the row and column index of the cell to which the formula belongs, the string representation of the formula, as well as the parser tree for quick evaluation.

FPSpreadsheet supports the majority of the formulas provided by the common spreadsheet applications. However, when reading a file created by these applications, there is always a chance that an unsupported formula is contained. To avoid crashing of fpspreadsheet, reading of formulas is disabled by default; the cell displays only the result of the formula written by the Office application. To activate reading of formulas add the element `boReadformulas` to the workbook's `Options` before opening the file. If an error occurs in this case the reader normally catches the exception, writes the exception message into the workbook's errorlog and continues reading. If you want reading to stop you must add the `boAbortReadingOnFormulaError` to the workbook `Options`.

Formulas can link to data in other sheets of the same workbook ("3d formulas") by applying the Excel syntax (see below). External links to spreadsheets in other files are not supported.

Calculation of formulas is normally not needed when a file is written by FPSpreadsheet for opening in an Office application because that automatically calculates the formula results. If the

same file, however, is openend by an application based on FPSpreadsheet the calculated cells would be empty because the formulas are not automatically calculated by default. To activate calculation of formulas before writing a spreadsheet to file you have to add the option boCalcBeforeSaving to the workbook's Options.

If FPSpreadshet is used in an interactive application (such as the *spready* demo found in the *examples* folder of the FPSpreadsheet installation) it is desirable to calculate formulas automatically whenever formula strings or cell values are changed by the user. This can be achieved by the option boAutoCalc in the workbook's Options.

The most general setting regarding formulas, therefore, is

```
MyWorkbook.Options := MyWorkbook.Options + [boReadFormulas, boCalcBeforeSaving, boAutoCalc];
```

Calculation of formulas can be triggered manually by calling the method CalcFormulas of the worksheet or workbook. The latter is absolutely required when the workbook contains 3d formulas where the result of one cell can affect cells in other sheets. If there are only within-sheet formulas then the worksheet's CalcFormulas is sufficient.

String formulas

String formulas are written in the same way as in the Office applications. The worksheet method for creating a string formula is WriteFormula:

```
var
  MyWorksheet: TsWorksheet;
//...
  MyWorksheet.WriteFormula(0, 1, '=ROUND(A1+B2+1.215,0)');
  // By default, use dot as decimal and comma as list separator!
```

A few notes on **syntax**:

- The leading = character which identifies formulas in the Office applications is not absolutely necessary here and can be dropped. The formula is stored in the cell record without it.
- The case of the formula name is ignored.
- Spaces can be added for better readability, but they will be lost when saving.
- Strings must be enclosed in double quotes.
- The corner points of a cell range must be separated by a colon (":"), e.g. A1:C3. Unordered ranges will be rearranged when the formula is parsed, i.e. C3:A1 becomes A1:C3.
- Links to other worksheets must follow the Excel syntax which separates the sheetname and cell address by a "!". An single cell, for example, can be linked by Sheet1!A1. An range of sheets must be placed before the cell or cell range, e.g. Sheet1:Sheet2!A1:C3. Note that the Open/LibreOffice syntax with a separating point and reference to the corner points of the 3d box (i.e., Sheet1.A1:Sheet2.C3) is not supported. Note also that the worksheet(s) to which the formula links must exist at the time when the formula is added; otherwise the link will be replaced by the error code #REF!, and the formula will not be usable even if the missing sheet is added later.
- Normally, floating point numbers must be entered with a dot as decimal separator, and a comma must be used to separate function arguments.
- Setting the optional parameter ALocalized of the worksheet methods WriteFormula to TRUE, however, allows to use localized decimal and list separators taken from the workbook's FormatSettings - see *spready* demo.

```
var
  MyWorksheet: TsWorksheet;
//...
  MyWorksheet.WriteFormula(0, 1, '=ROUND(A1+B2+1,215;0)', true);
  // Because of the "true" the formula parser accepts the comma as decimal and the
  // semicolon as list separator if the workbook's FormatSettings are set up like this.
```

Use the worksheet methods ReadFormula or ReadFormulaAsString to retrieve the string formula assigned to the cell. The pointer to the cell must be given as a parameter. The latter function accepts an additional (boolean) parameter ALocalized to use the decimal and list separators of the workbook's FormatSettings for creation of the formula string.

```
var
  MyWorksheet: TsWorksheet;
  cell: PCell;
//...
  cell := MyWorksheet.FindCell(0, 1);
  WriteLn('The formula in internal format is ', MyworkSheet.ReadFormula(cell));
  WriteLn('The localized formula is ', MyWorksheet.ReadFormulaAsString(cell, true));
//------------------------------------------------------------------------
// For the previous example, this will result in the following output
The formula in internal format is ROUND(A1+B2+1.215,0)
The localized formula is ROUND(A1+B2+1,215;0)
```

RPN formulas

At application level, string formulas are mainly used, and RPN formulas are of little practical importance. Therefore, documentation of RPN formulas has been removed from this main FPSpreadsheet wiki and can be found in the article "RPN Formulas in FPSpreadsheet".

Shared formulas and array formulas

- **Shared formulas** are only supported for reading (from Excel files).
- **Array formulas** are not supported, currently.

List of built-in formulas

FPSpreadsheet supports more than 80 built-in formulas. In order not to blow up this wiki page too much documentation of these formulas has been moved to the separate document "List of formulas".

To learn more about the functions available, look at file *testcases_calcrpnformula.inc* in the *tests* folder of the FPSpreadsheet installation where every function is included with at least one sample.

Extending FPSpreadsheet by user-defined formulas

Although the built-in formulas cover most of the applications there may be a need to access a formula which is available in the Office application, but not in FPSpreadsheet. For this reason, the library supports a registration mechanism which allows to add user-defined functions to the spreadsheets. This can be done by calling the procedure RegisterFunction from the unit *fpsExprParser*:

```
procedure RegisterFunction(const AName: ShortString; const AResultType: Char;
  const AParamTypes: String; const AExcelCode: Integer; ACallBack: TsExprFunctionCallBack); overload;

procedure RegisterFunction(const AName: ShortString; const AResultType: Char;
  const AParamTypes: String; const AExcelCode: Integer; ACallBack: TsExprEventCallBack); overload;
```

- AName specifies the name under which the function will be called in the spreadsheet. It must match the name of the formula in the Office application.
- AResultType is a character which identifies the data type of the function result:
  - 'F' - floating point number
  - 'I' - integer
  - 'D' - date/time
  - 'B' - boolean
  - 'S' - string

    - 'C' - cell address, e.g. 'A1'
    - 'R' - cell range address, e.g. 'A1:C3'
- AParamTypes is a string in which each character identifies the data type of the corresponding argument. In addition to the list shown above the following symbols can be used:
    - '?' - any type
    - '+' - must be the last character. It means that the preceding character is repeated indefinitely. This allows for an arbitrary argument count. Please note, however, that Excel supports only up to 30 arguments.
    - lowercase 'f', 'i', 'd', 'b', 's' indicate optional parameters of the type explained above. Of course, uppercase symbols cannot follow lower-case symbols.
- AExcelCode is the identifier of the function in xls files. See "OpenOffice Documentation of the Microsoft Excel File Format", section 3.11, for a list.
- ACallback identifies which function is called by FPSpreadsheet for calculation of the formula. It can either be a procedure or an event handler.

```
type
  TsExprFunctionCallBack = procedure (var Result: TsExpressionResult; const Args: TsExprParameterArray);
  TsExprFunctionEvent = procedure (var Result: TsExpressionResult; const Args: TsExprParameterArray) of object;
```

The TsExpressionResult is a variant record containing result or argument data of several types:

```
type
  TsResultType = (rtEmpty, rtBoolean, rtInteger, rtFloat, rtDateTime, rtString,
    rtCell, rtCellRange, rtError, rtAny);

  TsExpressionResult = record
    Worksheet       : TsWorksheet;
    ResString       : String;
    case ResultType : TsResultType of
      rtEmpty       : ();
      rtError       : (ResError       : TsErrorValue);
      rtBoolean     : (ResBoolean     : Boolean);
      rtInteger     : (ResInteger     : Int64);
      rtFloat       : (ResFloat       : TsExprFloat);
      rtDateTime    : (ResDateTime    : TDatetime);
      rtCell        : (ResRow, ResCol : Cardinal);
      rtCellRange   : (ResCellRange   : TsCellRange);
      rtString      : ();
  end;

  TsExprParameterArray = array of TsExpressionResult;
```

As an example we show here the code for the CONCATENATE() formula which joins two or more strings:

```
const
  INT_EXCEL_SHEET_FUNC_CONCATENATE = 336;
...
  RegisterFunction('CONCATENATE', 'S', 'S+', INT_EXCEL_SHEET_FUNC_CONCATENATE, @fpsCONCATENATE);

procedure fpsCONCATENATE(var Result: TsExpressionResult; const Args: TsExprParameterArray);
// CONCATENATE( text1, text2, ... text_n )
var
  s: String;
  i: Integer;
begin
  s := '';
  for i:=0 to Length(Args)-1 do
  begin
    if Args[i].ResultType = rtError then
    begin
      Result := ErrorResult(Args[i].ResError);
      exit;
    end;
    s := s + ArgToString(Args[i]);
    // "ArgToString" simplifies getting the string from a TsExpressionResult as
    // a string may be contained in the ResString and in the ResCell fields.
    // There is such a function for each basic data type.
  end;
  Result := StringResult(s);
  // "StringResult" stores the string s in the ResString field of the
  // TsExpressionResult and sets the ResultType to rtString.
  // There is such a function for each basic data type.
end;
```

There is a worked-out example (*demo_formula_func.pas*) in the folder *examples/other* of the FPSpreadsheet installation. In this demo, four financial functions (FV(), PV(), PMT(), RATE()) are added to FPSpreadsheet.

Unsupported formulas

Sometimes it is required to create files for the Office applications with formulas not supported by fpspreadsheet. This is possible to some extent when the workbook option boIgnoreFormulas is active. Then any arbitrary formula can be written to a cell, and the formula is not checked and not evaluated. The workbook can be written to an .ods or .xlsx file. The old xls file format cannot be used because the formula would have to be parsed to create the rpn formula needed.

In folder *examples/other* you can find the sample project *demo_ignore_formula* which creates an ods file with references to another data file - external references normally are not supported by fpspreadsheet, and thus the ignore-formulas workaround must be used. Note that this example does not work with xlsx because Excel writes information on the external links to separate xml files within the xlsx container.

Cell formatting

Number and date/time formats

Numbers and date/time values can be displayed in a variety of formats. In FPSpreadsheet this can be achieved in two ways:

- using **built-in number formats** by specifiying a value for the NumberFormat of the cell
- using a custom **format string**.

Number formats can be specified by these worksheet methods:

```
type
  TsWorksheet = class
  public
    // Set number formats alone
    function WriteNumberFormat(ARow, ACol: Cardinal; ANumberFormat: TsNumberFormat;
      const AFormatString: String = ''): PCell; overload;
    procedure WriteNumberFormat(ACell: PCell; ANumberFormat: TsNumberFormat;
      const AFormatString: String = ''); overload;

    function WriteNumberFormat(ARow, ACol: Cardinal; ANumFormat: TsNumberFormat;
      ADecimals: Integer; ACurrencySymbol: String = ''; APosCurrFormat: Integer = -1;
      ANegCurrFormat: Integer = -1): PCell; overload;
    procedure WriteNumberFormat(ACell: PCell; ANumFormat: TsNumberFormat;
      ADecimals: Integer; ACurrencySymbol: String = '';
      APosCurrFormat: Integer = -1; ANegCurrFormat: Integer = -1); overload;

    function WriteFractionFormat(ARow, ACol: Cardinal; AMixedFraction: Boolean;
      ANumeratorDigits, ADenominatorDigits: Integer): PCell; overload;
    procedure WriteFractionFormat(ACell: PCell; AMixedFraction: Boolean;
      ANumeratorDigits, ADenominatorDigits: Integer); overload;

    // Set date/time formats alone
    function WriteDateTimeFormat(ARow, ACol: Cardinal; ANumberFormat: TsNumberFormat;
```

```
      const AFormatString: String = ''): PCell; overload;
    procedure WriteDateTimeFormat(ACell: PCell; ANumberFormat: TsNumberFormat;
      const AFormatString: String = ''); overload;


    // Set cell values and number formats in one call

    // number values
    function WriteNumber(ARow, ACol: Cardinal; ANumber: double;
      AFormat: TsNumberFormat = nfGeneral; ADecimals: Byte = 2;
      ACurrencySymbol: String = ''): PCell; overload;
    procedure WriteNumber(ACell: PCell; ANumber: Double; AFormat: TsNumberFormat = nfGeneral;
      ADecimals: Byte = 2; ACurrencySymbol: String = ''); overload;
    function WriteNumber(ARow, ACol: Cardinal; ANumber: double;
      AFormat: TsNumberFormat; AFormatString: String): PCell; overload;
    procedure WriteNumber(ACell: PCell; ANumber: Double;
      AFormat: TsNumberFormat; AFormatString: String); overload;

    // date/time values
    function WriteDateTime(ARow, ACol: Cardinal; AValue: TDateTime;
      AFormat: TsNumberFormat = nfShortDateTime; AFormatStr: String = ''): PCell; overload;
    procedure WriteDateTime(ACell: PCell; AValue: TDateTime;
      AFormat: TsNumberFormat = nfShortDateTime; AFormatStr: String = ''); overload;
    function WriteDateTime(ARow, ACol: Cardinal; AValue: TDateTime;
      AFormatStr: String): PCell; overload;
    procedure WriteDateTime(ACell: PCell; AValue: TDateTime;
      AFormatStr: String); overload;

    // currency values
    function WriteCurrency(ARow, ACol: Cardinal; AValue: Double;
      AFormat: TsNumberFormat = nfCurrency; ADecimals: Integer = 2;
      ACurrencySymbol: String = '?'; APosCurrFormat: Integer = -1;
      ANegCurrFormat: Integer = -1): PCell; overload;
    procedure WriteCurrency(ACell: PCell; AValue: Double;
      AFormat: TsNumberFormat = nfCurrency; ADecimals: Integer = -1;
      ACurrencySymbol: String = '?'; APosCurrFormat: Integer = -1;
      ANegCurrFormat: Integer = -1); overload;
    function WriteCurrency(ARow, ACol: Cardinal; AValue: Double;
      AFormat: TsNumberFormat; AFormatString: String): PCell; overload;
    procedure WriteCurrency(ACell: PCell; AValue: Double;
      AFormat: TsNumberFormat; AFormatString: String); overload;
  ...
```

Built-in number formats

The **built-in formats** are defined by the enumeration `TsNumberFormat`. In spite of its name, the elements cover both number and date/time values:

```
type
  TsNumberFormat = (
    // general-purpose for all numbers
    nfGeneral,
    // numbers
    nfFixed, nfFixedTh, nfExp, nfPercentage, nfFraction,
    // currency
    nfCurrency, nfCurrencyRed,
    // dates and times
    nfShortDateTime, nfShortDate, nfLongDate, nfShortTime, nfLongTime,
    nfShortTimeAM, nfLongTimeAM, nfDayMonth, nfMonthYear, nfTimeInterval,
    // other (using format string)
    nfCustom);
```

- **nfGeneral** corresponds to the default formatting showing as many decimals as possible (the number `3.141592654` would be unchanged.)
- **nfFixed** limits the decimals. The number of decimal places has to be specified in the call to `WriteNumber`. Example: with 2 decimals, the number `3.141592654` becomes `3.14`.
- **nfFixedTh**: similar to `nfFixed`, but adds a thousand separator when the number is displayed as a string: The number `3.141592654` would remain like in the previous example because it is too small to show thousand separators. But the number `314159.2654` would become `314,159.26`, for 2 decimals.
- **nfExp** selects exponential presentation, i.e. splits off the exponent. The parameter `ADecimals` in `WriteNumber` determines how many decimal places are used. (The number `3.141592654` becomes `3.14E+00` in case of two decimals).
- **nfPercentage** displays the number as a percentage. This means that the value is multiplied by 100, and a percent sign is added. Again, specify in `ADecimals` how many decimal places are to be shown. (The number `3.141592654` is displayed as `314.92%`, in case of 2 decimals).
- **nfFraction** presents a number as a fraction. Details (mixed fraction?, maximum digit count for numerator or denominator) for can be specified in the worksheet method `WriteFractionFormat`.
- **nfCurrency** displays the number together with a currency symbol, and there are special rules how to display negative values (in brackets, or minus sign before or after the number). The FormatSettings of the workbook are used to define the currency sign and the way numbers are displayed (`FormatSettings.CurrencyString` for the currency symbol, `FormatSettings.CurrencyFormat` for positive, `FormatSettings.NegCurrFormat` for negative values). These settings can be overridden by specifying them in the call to `WriteCurrency` directly.
- **nfCurrendyRed** like `nfCurrency`, in addition negative values are displayed in red.
- **nfShortDateTime** presents the DateTimeValue of the cell in "short date/time format", i.e. days + two digit months + two digit year + hours + minutes, no seconds. The order of the date parts is taken from the workbook's `FormatSettings`. This applies also to the other date/time formats.
- **nfShortDate** creates a date string showing day + two-digit month + two-digit year
- **nfShortTime** creates a time string showing hours + minutes.
- **nfLongTime**, similar, but includes seconds as well
- **nfShortTimeAM**, similar to `nfShortTime`, but uses the AM/PM time format, i.e. hours go up to 12, and AM or PM is added to specify morning or evening/afternoon.
- **nfLongTimeAM**, like `nfShortTimeAM`, but includes seconds
- **nfTimeInterval**, like `nfLongTime`, but there can be more than 24 hours. The interval can also be expressed in minutes or seconds, if the format strings `[n]:ss`, or `[s]`, respectively, are used.
- **nfCustom** allows to specify a dedicated formatting string.

As already noted the workbook has a property `FormatSettings` which provides additional information to control the resulting formatting. This is essentially a copy of the `DefaultFormatSettings` declared in the *sysutils* unit (the elements `LongDateFormat` and `ShortDateFormat` are slightly modified to better match the default settings in the main spreadsheet applications). The main purpose of the `FormatSettings` is to add a simple way of localization to the number formats.

Number format strings

In addition to these pre-defined formats, more specialized formatting can be achieved by using the format constant `nfCustom` along with a dedicated **format string**. The format string is constructed according to Excel syntax which is close to the syntax of fpc's `FormatFloat` and `FormatDateTime` commands (accepted as well, see the online-help for these functions).

Here is a basic list of the symbols used:

Symbol	Meaning	Format string: Number --> Output string
General	Displays all decimal places of the number	'General': 1.2345678 --> '1.2345678'
0	Displays insignificant zeros if a number has less digits than there are zeros in the format. If used for decimal places then the number is rounded to as many decimal places as 0s are found.	'000': 1 --> '001' '0.0': 1 --> '1.0' '0.0': 1.2345678 --> '1.2'
*	Like "0" above, but does not display insignificant zeros.	'0.*': 1 --> '1.' '0.*': 1.2345678 --> '1.2'
?	Like "0" above, but insignificant zeros are replaced by space characters. Good for aligning decimal points and fractions	'??0': 1 --> ' 1' '0.0??': 1 --> '1.0 '
.	Decimal separator; will be replaced by the value used in the DecimalSeparator of the workbook's FormatSettings	'0.00': 8.9 --> '8.90'
,	Thousand separator; will be replaced by the value used in the ThousandSeparator of the workbook's FormatSettings. If at the end of a number formatting sequence the displayed value is divided by 1000.	'#,##0.00': 1200 --> '1,200.00' '0.00,': 1200 --> '1.20'
E+, e+	Displays a number in exponential format. The digits used for the exponent are defined by the number of zeros added the this symbol. The sign of the exponent is shown for positive and negative exponents.	'0.00E+00': 1200 --> 1.20E+03
E-, e-	Displays a number in exponential format. The digits used for the exponent are defined by the number of zeros added the this symbol. The sign of the exponent is shown only for negative exponents.	'0.00e-000': 1200 --> 1.20e003
%	Displays the number as a "percentage", i.e. the number is multiplied by 100 and a % sign is added.	'0.0%': 0.75 --> 75.0%
/	This symbol has two meanings: if the cell represents a "number" then the slash indicates formatting as fraction, place holders for numerator and denominator must follow. If the cell represents a "date/time" then the slash indicates the date separator which will be replaced by the DateSeparator of the workbook's FormatSettings	'#/#': 1.5 --> '3/2' '# #/#': 1.5 --> '1 1/2' '# #/16': 1.5 --> '1 8/16' also: see date/time examples below
:	Separator between hours, minutes and seconds of a date/time value. Will be replaced by the TimeSeparator of the workbook's FormatSettings.	see examples below
yyyy	Place holder for the year of a date/time value. The year is displayed as a four-digit number.	'yyyy/mm/dd': Jan 3, 2012 --> '2012-01-03' In this example, the DateSeparator is a dash character (-).
yy	Place holder for the year of a date/time value. The year is displayed as a two-digit number.	'yy/mm/dd': Jan 3, 2012 --> '12-01-03'
m	Place holder for the month of a date/time value. The month is shown as a number without extra digits. Please note that the m code can also be interpreted as the "minutes" of a time value (see below).	'yyyy/m/dd': Jan 3, 2012 --> '2012-1-03'
mm	Place holder for the month of a date/time value. The month is shown as a two-digit number, i.e. a leading zero is added for January to September. Please note that the mm code can also be interpreted as the "minutes" of a time value (see below).	'yyyy/mm/dd': Jan 3, 2012 --> '2012-01-03'
mmm	Place holder for the month of a date/time value. The month is displayed by its abbreviated name.	'yyyy/mmm/dd': Jan 3, 2012 --> '2012-Jan-03'
mmmm	Place holder for the month of a date/time value. The month is displayed by its full name.	'yyyy/mmm/dd': Jan 3, 2012 --> '2012-January-03'
d	Place holder for the day of a date/time value to be displayed as a number. The day is displayed as a simple number, without adding a leading zero.	'yyyy/mm/d': Jan 3, 2012 --> '2012-01-3'
dd	Place holder for the day of a date/time value to be displayed as a number. dd adds a leading zero to single-digit day numbers.	'yyyy/mm/dd': Jan 3, 2012 --> '2012-01-03'
ddd	Place holder for the day of a date/time value. The day is displayed as its abbreviated name.	'dddd, yyyy/mm/ddd': Jan 03, 2012 --> 'Tue 2012-01-03'
dddd	Place holder for the day of a date/time value. The day is displayed as its full name.	'dddd, yyyy/mmmm/dd': Jan 03, 2012 --> 'Tuesday 2012-01-03'
h	Place holder of the hour part of a date/time value. The hour is displayed as a simple number, without adding a leading zero.	'h:mm': 0.25 --> '6:00'
hh	Place holder of the hour part of a date/time value. The hour is displayed with a leading zero if the hour is less than 10.	'hh:mm': 0.25 --> '06:00'
[hh], or [h]	Displays elapsed time such that the hour part can become greater than 23	'[h]:mm': 1.25 --> '30:00'
m	Place holder of the minutes part of a date/time value. The minutes are shown as a simple number without adding a leading zero. Note that if the m codes are surrounded by date symbols (y, d) then they are interpreted as "month".	'h:m': 0.25 --> '6:0'
mm	Place holder of the minutes part of a date/time value. Single-digit minutes are displayed with a leading zero. Note that if the mm code is surrounded by date symbols (y, d) then it is interpreted as "month".	'h:mm': 0.25 --> '6:00'
[mm], or [m]	Displays elapsed time such that the minute part can become greater than 59	'[mm]:ss': 1.25 --> '1800:00'
s	Place holder of the seconds part of a date/time value. The seconds are displayed as a simple number, without adding a leading zero.	'hh:mm:s': 0.25 --> '06:00:0'
ss	Place holder of the seconds part of a date/time value. Single-digit seconds are displayed with a leading zero.	'hh:mm:ss': 0.25 --> '06:00:00'
[ss], or [s]	Displays elapsed time such that the seconds part can become greater than 59	'[ss]': 1.25 --> '108000'
AM/PM, am/pm, A/P, or a/p	Displays the time in the 12-hour format.	'hh:mm:ss AM/PM': 0.25 --> '6:00:00 AM'
"	The text enclosed by quotation marks is inserted into the formatted strings literally.	'yyyy"/"mm"/"dd': Jan 3, 2012 --> '2012/01/03' (i.e. the / is not replaced by the DateSeparator of the workbook).
\	The next character of the format string appears in the result string literally. The \ itself does not show up.	'yyyy\/mm\/dd': Jan 3, 2012 --> '2012/01/03'
;	A format string can contain up to three sections separated by the semicolon. The first section is used for positive numbers, the second section for negative numbers, and the third section for zero numbers. If the third section is missing then a zero value is formatted as specified in the first section. If the second section is missing as well then all values are formatted according to the first section.	'"#,##0"$"';-#,##0"$";"-"': 1200 --> '1,200$' -1200 --> '1,200$' 0 --> '-'

(, and)	Sometimes used for currency values to indicate negative numbers, instead of minus sign	`'#,##0"$";(#,##0)"$"'`: -1200 --> `'(1200)$'`
`[red]`	The formatted string is displayed in the specified color. Instead of `[red]`, you can use accordingly `[black]`, `[white]`, `[green]`, `[blue]`, `[magenta]`, `[yellow]`, or `[cyan]`. Often used to highlight negative currency values.	`'"$" #,##0.00;[red]("$" #,##0.00)'`: -1200 --> `'($ 1200.00)'`

A note on Excel dates

There is a discrepancy between FPSpreadsheet and Excel presentation of dates: Excel incorrectly assumes that the year 1900 was a leap year and thus allows for the date Febr 29 1900. FPSpreadsheet does the date calculation correctly. As consequence, dates before March 1 1900 are off by one! BTW, FPSpreadsheet handles this issue in the same way as LibreOffice Calc.

Colors

FPSpreadsheet supports colors for text, cell background, and cell borders. The basic EGA colors are declared in unit *fpstypes* as constants:

```
type
  TsColor = DWORD;

const
  scBlack       = $00000000;
  scWhite       = $00FFFFFF;
  scRed         = $000000FF;
  scGreen       = $0000FF00;
  scBlue        = $00FF0000;
  scYellow      = $0000FFFF;
  scMagenta     = $00FF00FF;
  scCyan        = $00FFFF00;
  scDarkRed     = $00000080;
  scDarkGreen   = $00008000;
  scDarkBlue    = $00800000;
  scOlive       = $00008080;
  scPurple      = $00800080;
  scTeal        = $00808000;
  scSilver      = $00C0C0C0;
  scGray        = $00808080;
  scGrey        = scGray; // redefine to allow different spelling

  // Identifier for undefined color
  scNotDefined  = $40000000;

  // Identifier for transparent color
  scTransparent = $20000000;
```

	A
1	black
2	white
3	red
4	green
5	blue
6	yellow
7	magenta
8	cyan
9	dark red
10	dark green
11	dark blue
12	olive
13	purple
14	teal
15	silver
16	gray

The `TsColor` represents the **rgb value** of a color, a single byte being used for the red, green, and blue components. The resulting number is in little endian notation, i.e. the red value comes first in memory: `$00BBGGRR`. (This is directly compatible with the color values as defined in the *graphics* unit.)

The high order byte is usually zero but is used internally to identify special color values, such as for undefined or transparent colors.

Note: In older versions of the library colors were defined as indexes into a color palette. **THIS IS NO LONGER WORKING.**

Unit *fpsutils* contains some useful functions for **modification of colors**:

- `function GetColorName(AColor: TsColor): String;`
  returns the name of the colors defined above, or a string showing the rgb components for other colors.
- `function HighContrastColor(AColor: TsColor): TsColor;`
  returns `scBlack` for a "bright", `scWhite` for a "dark" input color.
- `function TintedColor(AColor: TsColor; tint: Double): TsColor;`
  brightens or darkens a color by applying a factor `tint = -1..+1`, where `-1` means "100% darken", `+1` means "100% brighten", and `0` means "no change". The hue of the color is preserved.

Cell background

The cell background can be filled by predefined patterns which are identified by the record `TsFillPattern`:

```
type
  TsFillPattern = record
    Style: TsFillStyle;   // fill style pattern as defined below
    FgColor: TsColor;     // forground color of the fill pattern
    BgColor: TsColor;     // background color of the fill pattern
  end;

  TsFillStyle = (fsNoFill, fsSolidFill, fsGray75, fsGray50, fsGray25, fsGray12, fsGray6,
    fsStripeHor, fsStripeVert, fsStripeDiagUp, fsStripeDiagDown,
    fsThinStripeHor, fsThinStripeVert, fsThinStripeDiagUp, fsThinStripeDiagDown,
    fsHatchDiag, fsThinHatchDiag, fsThickHatchDiag, fsThinHatchHor);
```

- Use the worksheet method `WriteBackground` to assign a fill pattern to a specific cell. Besides the cell address, this method requires the type of the fill pattern (`TsFillStyle`), and the foreground and background colors as specified by their `TsColor` values.
- The fill pattern of a particular cell can be retrieved by calling the workbook method `ReadBackground`.
- The simplified method `WriteBackgroundColor` can be used to achieve a uniform background color.
- **Limitations:**
  - OpenDocument files support only uniform fills. The background color is a mixture of the foreground and background rgb components in a ratio defined by the fill pattern.
  - BIFF2 files support only a 12.5% black-and-white shaded pattern.

```
type
  TsWorksheet = class
  public
    function WriteBackground(ARow, ACol: Cardinal; AStyle: TsFillStyle; APatternColor, ABackgroundColor: TsColor): PCell; overload;
    procedure WriteBackground(ACell: PCell; AStyle: TsFillStyle; APatternColor, ABackgroundColor: TsColor); overload;

    function WriteBackgroundColor(ARow, ACol: Cardinal; AColor: TsColor): PCell; overload;
    procedure WriteBackgroundColor(ACell: PCell; AColor: TsColor); overload;

    function ReadBackground(ACell: PCell): TsFillPattern;
    function ReadBackgroundColor(ACell: PCell): TsColor; overload;
    // ...
  end;

var
  cell: PCell;
...
  // Example 1: Assign a pattern of thin, horizontal, yellow stripes on a blue background to empty cell A1 (row 0, column 0)
  MyWorksheet.WriteBackground(0, 0, fsThinStripeHor, scYellow, scBlue);

  // Example 2: Uniform gray background color of cell B1 (row 0, column 1) containing the number 3.14
  cell := MyWorksheet.WriteNumber(0, 1, 3.14);
  MyWorksheet.WriteBackgroundColor(cell, clSilver);
```

Cell borders

Cells can be emphasized by drawing border lines along their edges or diagonal lines. There are four borders plus two diagonals enumerated in the data type `TsCellBorder`:

```
type
  TsCellBorder = (cbNorth, cbWest, cbEast, cbSouth, cbDiagUp, cbDiagDown);
  TsCellBorders = set of TsCellBorder;
```

In order to show a border line add the corresponding border to the cell's set `Borders` (type `TsCellBorders`, see above). In this way, each cell edge can be handled separately. Use the worksheet method `WriteBorders` for this purpose. This example adds top and bottom borders to the edges A1 and B1:

```
  MyWorksheet.WriteBorders(0, 0, [cbNorth, cbSouth]);  // A1: row 0, column 0
  MyWorksheet.WriteBorders(0, 1, [cbNorth, cbSourth]); // B1: row 0, column 1
```

Lines usually are drawn as thin, solid, black lines. But it is possible to modify **line style** and **color** of each line. For this purpose, the cell provides an array of `TsCellBorderStyle` records:

```
type
  TsLineStyle = (lsThin, lsMedium, lsDashed, lsDotted, lsThick, lsDouble, lsHair,
    lsMediumDash, lsDashDot, lsMediumDashDot, lsDashDotDot, lsMediumDashDotDot, lsSlantDashDot);

  TsCellBorderStyle = record
    LineStyle: TsLineStyle;
    Color: TsColor;
  end;

  TsCellBorderStyles = array[TsCellBorder] of TsCellBorderStyle;

  TsWorksheet = class
  public
    function WriteBorderStyle(ARow, ACol: Cardinal; ABorder: TsCellBorder; AStyle: TsCellBorderStyle): PCell; overload;
    procedure WriteBorderStyle(ACell: PCell; ABorder: TsCellBorder; AStyle: TsCellBorderStyle); overload;

    function WriteBorderStyle(ARow, ACol: Cardinal; ABorder: TsCellBorder; ALineStyle: TsLineStyle; AColor: TsColor): PCell; overload;
    procedure WriteBorderStyle(ACell: PCell; ABorder: TsCellBorder; ALineStyle: TsLineStyle; AColor: TsColor); overload;

    function WriteBorderColor(ARow, ACol: Cardinal; ABorder: TsCellBorder; AColor: TsColor): PCell; overload;
    procedure WriteBorderColor(ACell: PCell; ABorder: TsCellBorder; AColor: TsColor): PCell; overload;

    function WriteBorderLineStyle(ARow, ACol: Cardinal; ABorder: TsCellBorder; ALineStyle: TsLineStyle): PCell; overload;
    procedure WriteBorderLineStyle(ACell: PCell; ABorder: TsCellBorder; ALineStyle: TsLineStyle): PCell; overload;

    function WriteBorderStyles(ARow, ACol: Cardinal; const AStyles: TsCellBorderStyles): PCell; overload;
    procedure WriteBorderStyles(ACell: PCell; const AStyles: TsCellBorderStyles); overload;

    function WriteBorders(ARow, ACol: Cardinal; ABorders: TsCellBorders): PCell; overload
    procedure WriteBorders(ACell: PCell; ABorders: TsCellBorders); overload

    function ReadCellBorders(ACell: PCell): TsCellBorders;
    function ReadCellBorderStyle(ACell: PCell; ABorder: TsCellBorder): TsCellBorderStyle;
    function ReadCellBorderStyles(ACell: PCell): TsCellBorderStyles;
    ...
  end;
```

The style of a given cell border can be specified by the following methods provided by the worksheet:

- **WriteBorderStyle** assigns a cell border style record to one border of the cell. There are two overloaded versions of this method: one takes an entire `TsCellBorderStyle` record, the other one takes the individual record elements.
- **WriteBorderColor** changes the color of a given border without affecting the line style of this border.
- **WriteBorderLineStyle** sets the line style of the border only, but leaves the color unchanged.
- **WriteBorderStyles** sets the border style of all borders of a given cell at once. Useful for copying border styles from one cell to other cells.

This example adds a thin black border to the top, and a thick blue border to the bottom of cells A1 and B1:

```
var
  cellA1, cellB1: PCell;
...
  cellA1 := MyWorksheet.WriteBorders(0, 0, [cbNorth, cbSouth]);    // cell A1: row 0, column 0
  MyWorksheet.WriteBorderStyle(cellA1, cbNorth, lsThin, scBlack);
  MyWorksheet.WriteBorderStyle(cellA1, cbSouth, lsThick, scBlue);

  cellB1 := MyWorksheet.WriteBorders(0, 1, [cbNorth, cbSouth]);    // cell B1: row 0, column 1
  MyWorksheet.WriteBorderStyles(cellB1, cellA1^.BorderStyles);     // copy all border styles from cell A1 to B1
```

Note: Diagonal lines are not supported by *sfExcel2* and *sfExcel5* - they are just omitted. *sfExcel8* and *sfOOXML* use the same linestyle and color for both diagonals; when writing the border style of the diagonal-up line is assumed to be valid also for the diagonal-down line. Some writers do not support all the line styles of the `TsLineStyle` enumeration. In this case, appropriate replacement line styles are used.

Fonts

The cell text can displayed in various fonts. For this purpose, the workbook provides a list of `TsFont` items:

```
type
  TsFont = class
    FontName: String;
    Size: Single;
    Style: TsFontStyles;
    Color: TsColor;
    Position: TsFontPosition;
  end;
```

- The **FontName** corresponds to the name of the font as used by the operational system. In Windows, an example would be "Times New Roman".
- The **FontSize** is given in "points", i.e. units 1/72 inch which are commonly used in Office applications.
- The **FontStyle** is a set of the items `fssBold`, `fssItalic`, `fssStrikeout`, and `fssUnderline` which form the enumeration type `TsFontStyle`. The "normal" font corresponds to an empty set.
- The **Color** determines the foreground color of the text characters given in rgb presentation as discussed above.
- The **Position** is either `fpNormal`, `fpSuperscript`, or `fpSubscript` and indicates whether the font size should be decreased by about 1/3 and the characters should be displaced up (superscript) or down (subscript).

Every cell is provided with an **index into the font list**.

In order to assign a particular font to a cell, use one of the following methods of `TsSpreadsheet`:

```
type
  TsSpreadsheet = class
  public
    function  WriteFont(ARow, ACol: Cardinal; const AFontName: String;
      AFontSize: Single; AFontStyle: TsFontStyles; AFontColor: TsColor): Integer; overload;
    procedure WriteFont(ARow, ACol: Cardinal; AFontIndex: Integer); overload;
    function WriteFontColor(ARow, ACol: Cardinal; AFontColor: TsColor): Integer;
    function WriteFontSize(ARow, ACol: Cardinal; ASize: Integer): Integer;
    function WriteFontStyle(ARow, ACol: Cardinal; AStyle: TsFontStyles): Integer;
    // plus: overloaded versions accepting a pointer to a cell record instead of the row and column index as parameter
    // ...
  end;
```

- **WriteFont** assigns a font to the cell. If the font does not yet exist in the font list a new entry is created. The function returns the index of the font in the font list. In addition, there is an overloaded version which only takes the font index as a parameter.
- **WriteFontColor** replaces the color of the font that is currently assigned to the cell by a new one. Again, a new font list item is created if the font with the new color does not yet exist. The function returns the index of the font in the list.
- **WriteFontSize** replaces the size of the currently used font of the cell.
- **WriteFontStyle** replaces the style (normal, bold, italic, etc.) of the currently used cell font.

The workbook's font list contains at least one item which is the **default font** for cells with unmodified fonts. By default, this is 10-point "Arial". Use the workbook method `SetDefaultFont` to assign a different font to the first list item.

The font at a given index of the font list can be looked up by calling the workbook function `GetFont`. The count of available fonts is returned by `GetFontCount`.

Here is an **example** which decreases the size of all 10-point "Arial" fonts to 9-point:

```
var
  i: Integer;
  font: TsFont;
begin
  for i := 0 to MyWorkbook.GetFontCount-1 do
  begin
    font := MyWorkbook.GetFont(i);
    if (font.FontName = 'Arial') and (font.Size = 10.0) then
      font.Size := 9.0;
  end;
end;
```

Rich-text formatting

In addition to using a specific font for each cell it is also possible to specify particular font attributes for individual characters or groups of characters in each cell text. Following the Excel notation, we call this feature **Rich-text formatting** (although is has nothing in common with the "rich-text" file format).

For this purpose, unit *fpstypes* declares the type `TsRichTextParams` which is an array of `TsRichTextParam` records:

```
type
  TsRichTextParam = record
    FontIndex: Integer;
    FirstIndex: Integer;
  end;

  TsRichTextParams = array of TsRichTextParam;
```

`FontIndex` refers to the index of the font in the workbook's FontList to be used for formatting of the characters beginning at the index `FirstIndex`. Being a string character index the `FirstIndex` is 1-based.

There are two ways to add "rich-text" formatting to a cell text:

- Embed corresponding HTML format codes into the cell text. This can be done using the method `WriteTextAsHTML` of the worksheet. In order to add the text "Area (m^2)" to cell A1, pass the following HTML-coded string to this function

```
  MyWorksheet.WriteTextAsHTML(0, 0, 'Area (m<sup>2</sup>');
```

- Alternatively, the standard text writing method, `WriteText` can be called with an additional parameter specifiying the rich-text formatting parameters to be used directly:

```
var
  richTextParams: TsRichTextParams;
  fnt: TsFont;
  cell: PCell;
begin
  SetLength(rtp, 2);
  cell := MyWorksheet.GetFont(0, 0);
  fnt := MyWorksheet.ReadCellFont(cell);
  richTextParams[0].FirstIndex := 8; // The superscript groups begins with "2" which is the (1-based) character #8 of the cell text.
  richTextParams[0].FontIndex := MyWorkbook.AddFont(fnt.FontName, fnt.Size, fnt.Style, fnt.Color, fpSuperscript);
  richTextParams[1].FirstIndex := 9; // Normal font again beginning with character #9.
  richTextParams[1].FontIndex := MyWorksheet.ReadCellFontIndex(0, 0);
  MyWorksheet.WriteUTF8Text(cell, 'Area (m2)', richTextParams);
end;
```

Use the worksheet method `DeleteRichTextParams` to remove rich-text formatting from a previously formatted cell.

> **Note:** If the cell is supposed to have a font different from the worksheet's default font then this font must be written to the cell **before** writing the rich-text formatted text. Otherwise the font in the unformatted regions will not be up-to-date.

Text rotation

Usually text is displayed in the cells horizontally. However, it is also possible to rotate it by 90 degrees in clockwise or counterclockwise directions. In addition, there is also an option to stack horizontal characters vertically above each other.

If you need this feature use the worksheet method `WriteTextRotation` and specify the text direction by an element of the enumeration type `TsTextRotation`:

```
type
  TsTextRotation = (trHorizontal, rt90DegreeClockwiseRotation,
    rt90DegreeCounterClockwiseRotation, rtStacked);

  TsWorksheet = class
  public
    function WriteTextRotation(ARow, ACol: Cardinal; ARotation: TsTextRotation): PCell; overload;
    procedure WriteTextRotation(ACell: PCell; ARotation: TsTextRotation); overload;

    function ReadTextRotation(ACell: PCell): TsTextRotation;
    // ...
  end;

  // example for counter-clockwise rotated text in cell A1:
  WriteTextRotation(0, 0, rt90DegreeCounterClockwiseRotation);
```

⚠**Warning:** Finer degrees of rotation which may be supported by some spreadsheet file formats are ignored.

Text alignment

By default, cell texts are aligned to the left and bottom edges of the cell, except for numbers which are right-aligned. This behavior can be changed by using the worksheet methods `WriteHorAlignment` and `WriteVertAlignment`:

```
type
  TsHorAlignment = (haDefault, haLeft, haCenter, haRight, haJustified, haDistributed, haFilled);
  TsVertAlignment = (vaDefault, vaTop, vaCenter, vaBottom);

  TsWorksheet = class
  public
    function WriteHorAlignment(ARow, ACol: Cardinal; AValue: TsHorAlignment): PCell; overload;
    procedure WriteHorAlignment(ACell: PCell; AValue: TsHorAlignment); overload;
    function ReadHorAlignment(ACell: PCell): TsHorAlignment;

    function WriteVertAlignment(ARow, ACol: Cardinal; AValue: TsVertAlignment): PCell; overload;
    procedure WriteVertAlignment(ACell: PCell; AValue: TsVertAlignment); overload;
    function ReadVertAlignment(ACell: PCell): TsVertAlignment;
    // ...
  end;

  // Example: Center the text in cell A1 both horizontally and vertically
  MyWorksheet.WriteHorAlignment(0, 0, haCenter);
  MyWorksheet.WriteVertAlignment(0, 0, vaCenter);
```

Explanation of special horizontal alignments:

- `haDistributed`: The cell text is word-wrapped and its spaces are expanded so that the entire width of the cell is occupied. The text is both left- and right-justified.
- `haJustified`: like `haDistributed`, expect for the last line which remains left-justified.
- `haFilled`: the text is repeated to fill the entire cell.

Word wrap

Text which is longer than the width of a cell can wrap into several lines by calling the method `WriteWordwrap` of the spreadsheet:

```
type
  TsWorksheet = class
  public
    function WriteWordwrap(ARow, ACol: Cardinal; AValue: Boolean): PCell; overload;
    procedure WriteWordwrap(ACell: PCell; AValue: Boolean); overload;
    function ReadWordwrap(ACell: PCell): Boolean;
    //...
  end;

  // Example: activate wordwrap in cell A1
  MyWorksheet.WriteWordwrap(0, 0, true);
```

Shrink-to-fit

The size of text which is longer than the width of a cell can be reduced such that the text fits into the cell. Call the worksheet method `WriteShrinkToFit` for this purpose:

```
type
  TsWorksheet = class
  public
    function WriteShrinkToFit(ARow, ACol: Cardinal; AValue: Boolean): PCell; overload;
    procedure WriteShrinkToFit(ACell: PCell; AValue: Boolean); overload;
    function ReadShrinkToFit(ACell: PCell): Boolean;
    ...
  end;

  // Example: activate shrink-to-fit in cell A1
  MyWorksheet.WriteShrinkToFit(0, 0, true);
```

This feature is available for reading/writing only for the Excel 8, Excel XLSX, Excel XML and LibreOffice ODS file formats.

Merged cells

Like the Office applications, FPSpreadsheet supports also to feature of merging cells to a single large cell which is often used as a common header above simlar columns. Simply call `MergeCells` and pass a parameter to specify the cell range to be merged, either an Excel range string (such as `A1:D5`), or the first and last rows and columns:

```
  MyWorksheet.MergeCells('A1:D5');
  // or: MyWorksheet.MergeCells(0, 0, 4, 3);  // first row, first column, last row, last column
```

The content and format displayed for a merged range is taken from the upper left corner of the range, cell `A1` in above example. This cell is called the `MergeBase` in the library. Except for this corner cell, there must not be any other cells in the range. If there are their contents and format will be hidden.

In order to break up a merged range back up into individual cells, use the command `Unmerge` and pass any cell that is within the merged range:

```
  MyWorksheet.UnmergeCells('B1');
  // or: MyWorksheet.UnmergeCells(0, 1);   // row, column of any cell within the range
```

Merged cells can be read from/written to all file formats except for `sfCSV`, `sfExcel2` and `sfExcel5` which do not support this feature natively.

The information which cells are merged is stored in an internal list. Unlike in earlier versions it is no longer possible to access the `MergeBase` from the cell directly. Use the following functions to extract information on merged cells:

```
var
  cell, base: PCell;
  r1,c1,r2,c2: Cardinal;
...
  cell := MyWorksheet.FindCell('B1');
  if MyWorksheet.IsMerged(cell) then
  begin
    WriteLn('Cell B1 is merged.');

    MyWorksheet.FindMergedRange(cell, r1, c1, r2, c2);
    WriteLn('The merged range is ' + GetCellRangeString(r1, c1, r2, c2));

    base := MyWorksheet.FindMergeBase(cell);
    WriteLn('The merge base is cell ' + GetCellString(base^.Row, base^.Col));
  end;
```

Cell protection

This is described in a separate section below.

Miscellaneous

Disable printing of specific cells

Specific cells can be disabled from printing:

```
type
  TsWorksheet = class
  public
    function WriteDoNotPrintCell(ARow, ACol: Cardinal; AValue: boolean): PCell; overload;
    procedure WriteDoNotPrintCell(ACell: PCell; AValue: Boolean); overload;
    function  ReadDoNotPrintCell(ACell: PCell): Boolean;
```

Additional data

Cell comments

Comments can be attached to any cell by calling

```
  MyWorksheet.WriteComment(0, 0, 'This is a comment for cell A1');
```

They are stored in an internal list of the worksheet. Use the corresponding worksheet methods to access comments:

- If you want to know whether a particular cell contains a comment call the worksheet method `HasComment(cell)`.
- For retrieving a cell comment use the method `ReadComment(cell)`, or its overloaded companion `ReadComment(ARow, ACol)`.
- The total number of comments can be retrieved from `worksheet.Comments.Count`.

Hyperlinks

Hyperlinks can be attached to cells in order to link cells to other documents or other cells in the same workbook. The general syntax for creating hyperlinks is

```
  procedure TWorksheet.WriteHyperlink(ARow, ACol: Cardinal; ATarget: String; ATooltip: String = '');
```

- The hyperlink target, passed as parameter `ATarget`, must be a fully qualified URI (Uniform resource identifier) (http://en.wikipedia.org/wiki/Uniform_Resource_Identifier) consisting of a protocol phrase (e.g., `http://`, `file:///`, `mailto:`, etc.) followed by specific information such as web URL, filename, or e-mail address and an optional bookmark identification separated by the character `'#'`. An exception are internal hyperlinks which enable to jump to a cell in the current workbook; they consist of the optional worksheet name and the cell address separated by the character `'!'`.
- The optional `Tooltip` parameter is evaluated by Excel to display it in a hint window if the mouse if above the hyperlink.


Note: Hyperlinks can be added to empty cells or to cells with content. In the latter case, the displayed content is not changed by the hyperlink; in the former case the cell is converted to a label cell showing the hyperlink target. Be aware that OpenOffice/LibreOffice only does accept hyperlinks with non-text cells.

Examples:

```
  // Open the web site www.google.com
  MyWorksheet.WriteText(0, 0, 'Open google');
  MyWorksheet.WriteHyperlink(0, 0, 'http://www.google.com');

  // Open the local file with the absolute path "C:\readme.txt" (assuming Windows)
  MyWorksheet.WriteHyperlink(1, 0, 'file:///c:\readme.txt');

  // Open the mail client to send a mail
  MyWorksheet.WriteText('Send mail');
  MyWorksheet.WriteHyperlink(3, 0, 'mailto:somebody@gmail.com?subject=Test');

  // Jump to a particular cell
  MyWorksheet.WriteText(5, 0, 'Jump to cell A10 on sheet2');
  MyWorksheet.WriteHyperlink(5, 0, '#Sheet2!A10');

  // Jump to cell A10 on the current sheet and display a popup hint
  MyWorksheet.WriteHyperlink(5, 0, '#A10', 'Go to cell A10');
```


Note: FPSpreadsheet does not "follow" the links, it only provides a mechanism to get access to the link data. TsWorksheetGrid from the *laz_fpspreadsheet_visual* package, however, fires the event `OnClickHyperlink` if a cell with an external hyperlink is clicked for fractions of a second. In the corresponding event handler, you can, for example, load a target spreadsheet, or open a web browser to display the linked web site. If the link is an internal link to another cell within the same workbook then the grid jumps to the related cell.

Defined Names

FPSpreadsheet allows to assign names to cells and cell ranges, simular to Excel and Calc. Since such "defined names" can be used in formulas the readability of formulas can be greatly improved.

A defined name is a simple class of type `TsDefinedName` with a string property `Name` and a record property `Range` of type `TsCellRange3D` which lists the row and column indices of the top/left and bottom/right corners of the cell range, as well as the worksheet indices of these two corner points.

```
type
  TsDefinedName = class
  ...
  public
    function RangeAsString(AWorkbook: TsBasicWorkbook): String;
    function RangeAsString_ODS(AWorkbook: TsBasicWorkbook): String;
    class function ValidName(AName: String): Boolean;
    property Name: String;
    property Range: TsCellRange3D;
  end;
```

A useful function may be the class function `ValidName` which checks the validity of a name:

- less than 256 characters long
- contains no spaces and no punctuation characters
- begins with a letter, with '_' or with '\'
- must not collide with cell references
- must be unique (case-insensitive)

The `TsDefinedName` methods `RangeAsString` and `RangeAsString_ODS` return the defined range's Range parameters as a string in Excel and ODS syntax, respectively.

The workbook and each worksheet have a list property `DefinedNames` which collects the defined names. The defined names stored in the workbook have **global** scope while those stored in worksheets are valid only **locally** within that sheet.

- There are overloaded methods Add for adding a single cell or a cell range in which the name as well as the cell/cell range parameters are provided. The methods return the index of the new item in the list.

```
var
  idxCell, idxRange: Integer;
...
  // Global defined name
  idxCell := workbook.DefinedNames.Add(AName: string; ASheetIndex: Integer; ARow, ACol: Cardinal);
  idxRange := workbook.DefinedNames.Add(AName: String; ASheetIndex1, ASheetIndex2, ARow1, ACol1, ARow2, ACol2: Cardinal);
  // Local defined name
  idxCell := worksheet.DefinedNames.Add(AName: string; ASheetIndex: Integer; ARow, ACol: Cardinal);
  idxRange := worksheet.DefinedNames.Add(AName: String; ASheetIndex1, ASheetIndex2, ARow1, ACol1, ARow2, ACol2: Cardinal);
```

- The default property `Items[AIndex]` allows to read a defined name from the list. The following code lists all global defined names:

```
var
  defName: TsDefinedName;
  i: Integer;
...
  for i := 0 to workbook.DefinedNames.Count-1 do
  begin
    defName := workbook.DefinedNames[i];
    WriteLn('Index ', i, ': Name="', defName.Name, '", Range=', defName.RangeAsString(workbook));
  end;
```

Note: Unlike Excel and ODS, local defined names are not allowed to have the same names as global defined names. Cross-worksheet local defined names are not allowed either.

Images

FPSpreadsheet supports embedding of images in worksheets. Use one of the worksheet methods `WriteImage()` to add an image to the worksheet:

```
  MyWorksheet.WriteImage(row, col, filename, offsetX, offsetY, scaleX, scaleY);
  MyWorksheet.WriteImage(row, col, stream, offsetX, offsetY, scaleX, scaleY);
  MyWorksheet.WriteImage(row, col, imageindex, offsetX, offsetY, scaleX, scaleY);
```

The upper/left corner of the image is placed at the upper/left corner of the cell in the specified row and column. The floating point parameters `offsetX`, `offsetY`, `scaleX` and `scaleY` are optional: they define an offset of this image anchor point from the cell corner and a magnification factor. The path to the image file is given as parameter `filename`. Alternatively, overloaded versions can be used which accept a stream in place of the file name or an image index in the workbook's EmbeddedObj list - use `MyWorkbook.FindEmbeddedObj(filename)` to get this index for a previously loaded image file.

Note that FPSpreadsheet needs to know the image type for successfull picture import. Currently, the types *png*, *jpg*, *tiff*, *bmp*, *gif*, *svg*, *wmf*, *emf*, and *pcx* are supported (Excel2007 cannot read imported *svg* and *pcx* images). Other formats can be registered by writing a function which determines the image size and pixel density, and by registering the new format using the procedure `RegisterImageType` - see unit *fpsImages* for examples:

```
type
  TsImageType = integer;
  TGetImageSizeFunc = function (AStream: TStream; out AWidth, AHeight: DWord; out dpiX, dpiY: Double): Boolean;

function RegisterImageType(AMimeType, AExtension: String; AGetImageSize: TGetImageSizeFunc): TsImageType;
```

Due to differences in row height and column width calculation between FPSpreadsheet and Office applications it is not possible to position images correctly. If exact image positions are important you should follow these rules:

- Predefine the widths of all columns at least up to the one containing the right edge of the image.
- Predefine the heights of all rows at least up to the one containing the lower edge of the image.
- If the workbook is to be saved in OpenDocument format add the image after changing column widths and row heights because ods anchors the image to the sheet, not to the cell (like Excel and FPSpreadsheet).
- If the exact size of the image is important make sure that it fits into a single cell.

Sorting

Cells in a worksheet can be sorted for a variety of criteria by calling the `Sort` method of the worksheet. This method takes a `TsSortParams` record and the edges of the cell rectangle to be sorted as parameters; in an overloaded version, the cell rectangle can also be specified by means of an Excel-type range string (e.g. `'A1:G10'`):

```
type
  TsWorksheet = class
    // ...
    procedure Sort(const ASortParams: TsSortParams;
      ARowFrom, AColFrom, ARowTo, AColTo: Cardinal); overload;
    procedure Sort(ASortParams: TsSortParams; ARange: String); overload;
    // ...
  end;
```

The sorting criteria are defined by a record of type `TsSortParams`:

```
type
  TsSortParams = record
    SortByCols: Boolean;
    Priority: TsSortPriority;  // spNumAlpha ("Numbers first"), or spAlphaNum ("Text first")
    Keys: TsSortKeys;
  end;

  TsSortKey = record
    ColRowIndex: Integer;
    Options: TsSortOptions;    // set of [spDescending, spCaseInsensitive]
  end;
```

- The boolean value `SortByCols` determines wether sorting occurs along columns (`true`) or rows (`false`). The `ColRowIndex` specified in the sorting keys, accordingly, corresponds to a column or row index, respectively (see below).
- `Priority` determines in mixed content cell ranges whether an ascending sort puts numerical values in front of text values or not. Empty cells are always moved to the end of the sorted column or row. In Excel, the priority is "numbers first" (`spNumAlpha`).
- The array `Keys` specifies multiple sorting parameters. They consist of the index of the column or row to be sorted (`ColRowIndex`) and a set of `Options` for sorting direction (`spoDescending`) and character case (`spCaseInsensitive`). If `Options` is empty, cell comparison is case-sensitive, and cells are arranged in ascending order. If two cells are found to be "equal" on the basis of the first key (`sortParams.Keys[0]`) comparison proceeds with the next conditions in the `Keys` array until a difference is found or all conditions are used up.

`InitSortParams` is a handy utility to initialize the sorting parameters:

```
function InitSortParams(ASortByCols: Boolean = true; ANumSortKeys: Integer = 1;
  ASortPriority: TsSortPriority = spNumAlpha): TsSortParams;
```

The next code fragment shows a typical sorting call:

```
var
  sortParams: TsSortParams;
begin
  sortParams := InitSortParams(true, 2);  // sorting along columns, 2 sorting keys

  // primary sorting key: column 3, ascending, case-insensitive
  sortParams.Keys[0].ColRowIndex := 3;
  sortParams.Keys[0].Options := [ssoCaseInsensitive];

  // secondary sorting key: colum 1, descending
  sortParams.Keys[1].ColRowIndex := 1;
  sortParams.Keys[1].Options := [ssoDescending];

  // The sorted block extends between cells A1 (row=0, col=0) and F10 (row=9, col=5)
  MyWorksheet.Sort(sortParams, 0, 0, 9, 5);
  // or: MyWorksheet.Sort(sortParams, 'A1:F10');
end;
```

Searching and replacing

Unit *fpsSearch* implements a **search engine** which can be used to look for specific cell content within a workbook, or to replace the found cell content by some other string.

Example:

```
uses
  fpsTypes, fpSpreadsheet, fpsUtils, fpsSearch, fpsAllFormats;
var
  MyWorkbook: TsWorkbook;
  foundWorksheet: TsWorksheet;
  foundRow, foundCol: Cardinal;
  MySearchParams: TsSearchParams;
begin
  MyWorkbook := TsWorkbook.Create;
  try
    MyWorkbook.ReadFromFile(AFileName);

    // Specify search criteria
    MySearchParams.SearchText := 'Hallo';
    MySearchParams.Options := [soEntireDocument];
    MySearchParams.Within := swWorkbook;

    // or: MySearchParaams := InitSearchParams('Hallo', [soEntireDocument], swWorkbook);

    // Create search engine and execute search
    with TsSearchEngine.Create(MyWorkbook) do begin
      if FindFirst(MySearchParams, foundWorksheet, foundRow, foundCol) then begin
        WriteLn('First "', MySearchparams.SearchText, '" found in cell ', GetCellString(foundRow, foundCol), ' of worksheet ', foundWorksheet.Name);
        while FindNext(MySeachParams, foundWorksheet, foundRow, foundCol) do
          WriteLn('Next "', MySearchParams.SearchText, '" found in cell ', GetCellString(foundRow, foundCol), ' of worksheet ', foundWorksheet.Name);
      end;
      Free;
    end;
  finally
    MyWorkbook.Free;
  end;
end;
```

The search engine provides two methods for **searching**: `FindFirst` and `FindNext`. They are very similar, they only differ in where the search begins. In case of `FindFirst`, the starting cell is determined from the `Options` described below. In case of `FindNext` the search begins at the cell adjacent to the previously found cell. Both methods return the worksheet and row and column indexes of the cell in which the search text is found. If the search is not successful then the function result is `FALSE`, and the `foundWorksheet` is nil. It is clear that the `foundWorksheet` cannot be used for anything else while the search is running.

The record `TsSearchParams` specifies the criteria used for searching:

```
type
  TsSearchParams = record
    SearchText: String;
    Options: TsSearchOptions;
    Within: TsSearchWithin;
  end;
```

Besides the text to be searched (`SearchText`) it provides a set of options to narrow the search:

- `soCompareEntireCell`: Compares the `SearchText` with the entire cell content. If not contained in the `Options` then the cell text is compared only partially.
- `soMatchCase`: Perform a case-sensitive search
- `soRegularExpr`: The `SearchText` is considered as a regular expression which is analyzed by the FCL class `TRegExpr`.
- `soAlongRows`: The search engine proceeds first along the rows. If not contained in the `Options` then the search proceeds along the columns.
- `soBackward`: The search begins at the end of the document, or runs backward from the active cell. If not contained in the `Options` then the search starts at the beginning of the document, or runs forward from the active cell.
- `soWrapDocument`: If a search has reached the end of the document the search is resumed at its beginning (or vice versa, if `soBackward` is used).
- `soEntireDocument`: Search begins at the first cell (or last cell if `soBackward` is used). If not contained in the `Options` then the search begins at the active cell of the worksheet. Ignored by `FindNext`.

The record field `Within` identifies the part of the spreadsheet to be searched:

- `swWorkbook`: The entire workbook is searched. If the search phrase is not found on the first worksheet (or last worksheet if `soBackward` is used) then the search continues with the next (previous) sheet.
- `swWorksheet`: The search is limited to the currently active worksheet
- `swColumn`: Search is restricted to the column of the active cell
- `swRow`: Search is restricted to the row of the active cell.

The search params record can be initialized by calling `InitSearchParams` (in unit *fpsutils*) with the record elements as optional parameters. Use the methods `Workbook.ActiveWorksheet` and `Worksheet.SelectCell(ARow, ACol)` to define the active worksheet and the position of the active cell, respectively, if needed by the search.

In addition to searching the search engine can also be used for **replacing** the found text by another string. Call the functions `ReplaceFirst` or `ReplaceNext` for this purpose. They act like their `FindXXXX` counterparts, therefore, they require a `TsSearchParams` record to specify the search criteria. But in addition to searching, these functions also perform the text replacement according to the specification in a `TsReplaceParams` record:

```
type
  TsReplaceParams = record
    ReplaceText: String;
    Options: TsReplaceOptions;
  end;
```

The `ReplaceText` identifies the string which will replace the found text pattern. The `Options` define a set of criteria how the replacement is done:

- `roReplaceEntirecell`: Replaces the entire cell text by the `ReplaceText`. If not contained in the `Options` then only the part matching the `SearchText` is replaced.
- `roReplaceAll`: Performs the replacement in all found cells (i.e., simply call `ReplaceFirst` to replace all automatically).
- `roConfirm`: Calls an event handler for the `OnConfirmReplacement` event in which the user must specify whether the replacement is to be performed or not. Note that this event

handler is mandatory if `roConfirm` is set.

Use the function `InitReplaceParams` (in unit *fpsutils*) to initialize the replace parameters record with the provided (but optional) values.

Column and row operations

The worksheet provides these methods for inserting, deleting, hiding or unhiding columns and rows:

```
type
  TsWorksheet = class
  ...
    procedure DeleteCol(ACol: Cardinal);
    procedure DeleteRow(ARow: Cardinal);

    procedure InsertCol(ACol: Cardinal);
    procedure InsertRow(ARow: Cardinal);

    procedure RemoveCol(ACol: Cardinal);
    procedure RemoveRow(ARow: Cardinal);

    procedure RemoveAllCols;
    procedure RemoveAllRows

    procedure HideCol(ACol: Cardinal);
    procedure HideRow(ARow: Cardinal);

    procedure ShowCol(ACol: Cardinal);
    procedure ShowRow(ARow: Cardinal);

    function ColHidden(ACol: Cardinal): Boolean;
    function RowHidden(ARow: Cardinal): Boolean;
```

- When a column or row is **deleted** by `DeleteCol` or `DeleteRow`, any data assigned to this column or row are removed, i., cells, comments, hyperlinks, TCol or TRow records. Data at the right of or below the deleted column/row move to the left or up.
- `RemoveCol` and `RemoveRow`, in contract, **remove** only the column or row record, i.e. reset column width and row height to their default values. Cell, comment, and hyperlink data are not affected.
- `RemoveAllCols` **removes all** column records, i.e. resets all column widths; `RemoveAllRows` does the same with the row records and row heights.
- A column or row is **inserted** before the index specified as parameter of the `InsertXXX` method.

Page layout

General

So far, FPSpreadsheet does not support printing of worksheets, but the Office applications do, and they provide a section of information in their files for this purpose. In FPSpreadsheets this information is available in the `TsPageLayout` class which belongs to the `TsWorksheet` data structure. Its properties and methods combine the most important features from the Excel and OpenDocument worlds.

```
type
  TsPageOrientation = (spoPortrait, spoLandscape);

  TsPrintOption = (poPrintGridLines, poPrintHeaders, poPrintPagesByRows,
    poMonochrome, poDraftQuality, poPrintCellComments, poDefaultOrientation,
    poUseStartPageNumber, poCommentsAtEnd, poHorCentered, poVertCentered,
    poDifferentOddEven, poDifferentFirst, poFitPages);

  TsPrintOptions = set of TsPrintOption;

  TsHeaderFooterSectionIndex = (hfsLeft, hfsCenter, hfsRight);

  TsCellRange = record
    Row1, Col1, Row2, Col2: Cardinal;
  end;

  TsPageLayout = class
  ...
  public
    ...
    { Methods }
    // embedded header/footer images
    procedure AddHeaderImage(AHeaderIndex: Integer;
      ASection: TsHeaderFooterSectionIndex; const AFilename: String);
    procedure AddFooterImage(AFooterIndex: Integer;
      ASection: TsHeaderFooterSectionIndex; const AFilename: String);
    procedure GetImageSections(out AHeaderTags, AFooterTags: String);
    function HasHeaderFooterImages: Boolean;

    // Repeated rows and columns
    function HasRepeatedCols: Boolean;
    function HasRepeatedRows: Boolean;
    procedure SetRepeatedCols(AFirstCol, ALastCol: Cardinal);
    procedure SetRepeatedRows(AFirstRow, ALastRow: Cardinal);

    // print ranges
    function AddPrintRange(ARow1, ACol1, ARow2, ACol2: Cardinal): Integer; overload;
    function AddPrintRange(const ARange: TsCellRange): Integer; overload;
    function GetPrintRange(AIndex: Integer): TsCellRange;
    function NumPrintRanges: Integer;
    procedure RemovePrintRange(AIndex: Integer);

    { Properties }
    property Orientation: TsPageOrientation read FOrientation write FOrientation;
    property PageWidth: Double read FPageWidth write FPageWidth;
    property PageHeight: Double read FPageHeight write FPageHeight;
    property LeftMargin: Double read FLeftMargin write FLeftMargin;
    property RightMargin: Double read FRightMargin write FRightMargin;
    property TopMargin: Double read FTopMargin write FTopMargin;
    property BottomMargin: Double read FBottomMargin write FBottomMargin;
    property HeaderMargin: Double read FHeaderMargin write FHeaderMargin;
    property FooterMargin: Double read FFooterMargin write FFooterMargin;
    property StartPageNumber: Integer read FStartPageNumber write SetStartPageNumber;
    property ScalingFactor: Integer read FScalingFactor write SetScalingFactor;
    property FitHeightToPages: Integer read FFitHeightToPages write SetFitHeightToPages;
    property FitWidthToPages: Integer read FFitWidthToPages write SetFitWidthToPages;
    property Copies: Integer read FCopies write FCopies;
    property Options: TsPrintOptions read FOptions write FOptions;
    property Headers[AIndex: Integer]: String read GetHeaders write SetHeaders;
    property Footers[AIndex: Integer]: String read GetFooters write SetFooters;
    property RepeatedCols: TsRowColRange read FRepeatedCols;
    property RepeatedRows: TsRowColRange read FRepeatedRows;
    property PrintRange[AIndex: Integer]: TsCellRange read GetPrintRange;
    property FooterImages[ASection: TsHeaderFooterSectionIndex]: TsHeaderFooterImage read GetFooterImages;
    property HeaderImages[ASection: TsHeaderFooterSectionIndex]: TsHeaderFooterImage read GetHeaderImages;
  end;

  TsWorksheet = class
  ...
  public
    property PageLayout: TsPageLayout;
    ...
  end;
```

Note: The fact that the `PageLayout` belongs to the worksheet indicates that there can be several page layouts in the same workbook, one for each worksheet.

- **Orientation** defines the orientation of the printed paper, either portrait or landscape.
- **Page width** and **page height** refer to the standard orientation of the paper, usually portrait orientation.
- **Left**, **top**, **right** and **bottom margins** are self-explanatory and are given in millimeters.
- **HeaderMargin** is understood - like in Excel - as the distance between the paper top edge and the top of the header, and **TopMargin** correspondingly is the distance between the top paper edge and the top of the first table row, i.e. if the header contains several line breaks it can reach into the the table part of the print-out. This is different from OpenDocument files where the header can grow accordingly.
- **StartPageNumber** should be altered if the print-out should not begin with page 1. This setting requires to add the option `poUseStartPageNumber` to the PageLayout's `Options` - but this is normally done automatically.
- The **ScalingFactor** is given in percent and can be used to reduce the number of printed pages. Modifying this property clear the option `poFitToPages` from the PageLayout's `Options`.
- Alternatively to ScalingFactor, you can also use **FitHeightToPages** or **FitWidthToPages**. The option `poFitToPages` must be active in order to override the `ScalingFactor`setting. `FitHeightToPages` specifies the number of pages onto which the entire height of the printed worksheet should fit. Accordingly, `FitWidthToPages` can be used to define the number of pages on which the entire width of the worksheet has to fit. The value `0` has the special meaning of "use as many pages as needed". In this way, the setting "Fit all columns on one page" of Excel, for example, can be achieved by this code:

```
MyWorksheet.PageLayout.Options := MyWorksheet.PageLayout.Options + [poFitPages];
MyWorksheet.PageLayout.FitWidthToPages := 1;     // all columns on one page width
MyWorksheet.PageLayout.FitHeightToPages := 0;    // use as many pages as needed
```

- **Header rows and columns repeated on every printed page** can be defined by the `RepeatedCols` and `RepeatedRows` records; their elements `FirstIndex` and `LastIndex` refer to the indexes of the first and last column or row, respectively, to be repeated. Use the methods `SetRepeatedCols` and `SetRepeatedRows` to define these numbers. Note that the second parameter for the last index can be omitted to use only a single header row or column.
- **Print ranges** or **print areas** (using Excel terminology) can be used to restrict printing only to a range of cells. Use the methods `AddPrintRange` to define a cell range for printing: specify the indexes of the left column, top row, right column and bottom row of the range to be printed. A worksheet can contain several print ranges.
- **Copies** specifies how often the worksheet will be printed.
- The **Options** define further printing properties, their names are self-explaining. They were defined according to Excel files, some of them do not exist in ODS files and are ignored there.

Headers and footers

Header and footer texts can be composed of left-aligned, centered and right-aligned strings. Add the symbol `&L` to indicate that the following string is to be printed as the left-aligned part; use `&C` accordingly for the centered and `&R` for the right-aligned parts. There are other symbols which will be replaced by their counterparts during printing:

- `&L`: begins the **left-aligned section** of a header or a footer text definition
- `&C`: begins the **centered section** of a header or a footer text definition
- `&R`: begins the **right-aligned section** of a header or a footer text definition
- `&P`: **page number**
- `&N`: **page count**
- `&D`: current **date** of printing
- `&T`: current **time** of printing
- `&A`: **worksheet name**
- `&F`: **file name** without path
- `&Z`: **file path** without file name
- `&G`: embedded **image** - use the methods `AddHeaderImage` or `AddFooterImage` to specify the image file; this also appends the `&G` to the other codes of the current header/footer section. Note that not all image types known by the Office application may be accepted. Currently the image can be *jpeg*, *png*, *gif*, *bmp*, *tiff*, *pcx*, *svg*, *wmf* or *emf*.
- `&B`: **bold** on/off
- `&I`: **italic** on/off
- `&U`: **underlining** on/off
- `&E`: **double-underlining** on/off
- `&S`: **strike-out** on/off
- `&H`: **shadow** on/off
- `&O`: **outline** on/off
- `&X`: **superscript** on/off
- `&Y`: **subscript** on/off
- `&"font"`: begin using of the **font** with the specified name, e.g. `&"Arial"`
- `&number`: begin using of the specified **font size** (in points), e.g. `&16`
- `&Krrggbb`: switch to the **font color** specified to the binary value of the specified color, e.g. use `&KFF0000` for red.

The arrays `Headers[]`/`Footers[]` provide space for usage of three different headers or footers:

- `Headers[0]` refers to the header used on the **first page** only, similarly for `Footers[0]`. Instead of index `0` you can use the constant `HEADER_FOOTER_INDEX_FIRST`. Leave this string empty if there is no special first-page header/footer.
- `Headers[1]` refers to the header on pages with **odd page numbers**, similarly for `Footers[1]`. Instead of index `1` you may want to use the constant `HEADER_FOOTER_INDEX_ODD`.
- `Headers[2]` refers to the header on lages with **even page nubmers**, similarly for `Footers[2]`. Instead of index `2` you may want to use the constant `HEADER_FOOTER_INDEX_EVEN`.

Leave the strings at index 0 and 2 empty if the print-out should always have the same header/footer. You can use the constant `HEADER_FOOTER_INDEX_ALL` for better clarity. Example:

```
MyWorksheet.PageLayout.Headers[HEADER_FOOTER_INDEX_ALL] := '&C&D &T';          // centered "date time" on all pages as header
MyWorksheet.PageLayout.Footers[HEADER_FOOTER_INDEX_ODD] := '&RPage &P of &N';  // right-aligned "Page .. of ..." on odd pages as footer
MyWorksheet.PageLayout.Footers[HEADER_FOOTER_INDEX_EVEN] := '&LPage &P of &N'; // dto, but left-aligned on even pages
```

Protection

In the Office applications, workbooks can be protected from unintentional changes by the user. fpspreadsheet is able to read and write the data structures related to protection, but does not enforce them. This means, for example, that cells can be modified by the user although the worksheet is specified as being locked.

Protection is handled at three levels: workbook protection, worksheet protection and cell protection.

Workbook protection

TsWorkbook contains a set of workbook or document protection options:

- bpLockRevision: specifies that the workbook is locked for revisions
- bpLockStructure: if this option is set then worksheets in the workbook cannot be moved, deleted, hidden, unhidden, or renamed, and new worksheets cannot be inserted.
- bpLockWindows: indicates that the workbook windows in the Office application are locked. Windows are the same size and position each time the workbook is opened by the Office application.

In relation to workbook protection is the worksheet option soPanesProtection which prevents the panes of a worksheet from being modified if the workbook is protected.

Depending on the file format, only some of these options might be supported. In these cases, the non-supported options are commonly accepted default values.

Office applications are able to encrypt their spreadsheet files. The readers of FPSpreadsheet basically are not able to open these files. However, there exists a related package, **laz_fpspreadsheet_crypto.lpk**, which has access to decryption routines and is able to read xlsx and ods files. These readers are in the units *xlsxOOXML_crypto* and *fpsOpendocument_crypto*, respectively. Adding them to the uses clause of the application installs user-provided readers with FormatID sfidOOXML_crypto and sfidOpenDocument_crypto, respectively. If the password is known it can be specified as a parameter to the workbook's LoadFromFile/ReadFromStream methods. If no password is specified the workbook fires the event OnQueryPassword in which the user can provide the password, e.g. like this:

```
function TForm1.QueryPasswordHandler: String;
begin
  Result := InputBox('Password required', 'Password', '');
end;

procedure TForm1.LoadWorkbook(AFileName: String);
var
  workbook: TsWorkbook;
begin
  workbook := TsWorkbook.Create;
  try
    workbook.OnQueryPassword := @QueryPasswordHandler;
    workbook.ReadFromFile(AFileName);
    ...
  finally
    workbook.Free;
  end;
end;

// or

procedure TForm1.LoadWorkbookWithPassword(AFileName, APassword: String);
var
  workbook: TsWorkbook;
begin
  workbook := TsWorkbook.Create;
  try
    workbook.ReadFromFile(AFileName, APassword);
    ...
```

Worksheet protection

TsWorksheet houses a similar set of protection options. Whenever an option is included in the set Protection of the workbook the corresponding action is not allowed and locked:

- spCells: the cells in the sheet are protected. It depends on the level of cell protection whether a particular cell can be changed or not. By default, no cell can be changed.
- spDeleteColumns: deleting of columns is not be allowed
- spDeleteRows: it is not possible to delete rows
- spFormatCells: formatting of cells is not allowed
- spFormatColumns: columns cannot be formatted.
- spFormatRows: rows cannot be formatted
- spInsertColumns: it is not allowed to insert columns
- spInsertRows: rows cannot be inserted
- spInsertHyperlinks: it is not possible to insert new hyperlinks
- spSort: the worksheet is not allowed to be sorted.
- spSelectLockedCells: Cells which are locked cannot be selected any more.
- spSelectUnlockedCells: Even cells which are unlocked cannot be selected. Together with spSelectLockedCells this means that the selection in the worksheet is frozen.

These levels of protection become active if the option soProtected is added to the worksheet's Options, or by calling the worksheet method Protect(true).

Cell protection

Cell protection becomes active when the worksheet protection is enabled. It is controlled by a set of TsCellProtection elements which belong to the cell format record:

- cpLockCell: This option determines whether cell content can be modified by the user. Since it is on by default cells of a protected worksheet normally cannot be edited. In order to unlock some cells for user input the option cpLockCell must be removed from the protection of these cells.
- cpHideFormulas: prevents formulas from being shown in the Office application.

Cell protection can be changed by calling the worksheet method WriteCellProtection. Conversely, ReadCellProtection can be used to retrieve the protection state of a particular cell:

```
// query and modifiy the protection state of cell A1 (row=0, col=0)
var
  cell: PCell;
  cellprot: TsCellProtections;
...
  // Find the cell
  cell := worksheet.FindCell(0, 0);
  // query cell protection
  cellprot := worksheet.ReadCellProtection(cell);
  // Unlock the cell for editing, don't change the visibility of formulas
  worksheet.WriteCellProtection(cell, cellprot - [cpLockCell]);
  // Hide formula of the cell and unlock the cell.
  worksheet.WriteCellProtection(cell, [cpHideFormulas]);
  // Activate protection
  worksheet.Protect(true);
```

Passwords

Workbook and worksheet protection can be secured by passwords. Note that these passwords do not encrypt the file (except for workbook protection in Excel 2007). In the Office applications the user must enter this password to turn off protection or to change protection items. The encrypted password is stored in the CryptoInfo record of the wordbook and the worksheets, respectively:

```
type
  TsCryptoInfo = record
    PasswordHash: String;
    Algorithm: TsCryptoAlgorithm;  // caExcel, caSHA1, caSHA256, etc.
    SaltValue: String;
    SpinCount: Integer;
  end;
```

⚠ **Warning:** FPSpreadsheet does not perform any hashing calculations, the CryptoInfo record is just passed through from reading to writing. This causes problems when different

file formats are involved in reading and writing. It is attempted to detect **incompatible combinations**. In these cases, the **password protection is removed**, and an error is logged by the workbook.

Loading and saving

Adding new file formats

FPSpreadsheet is open to any spreadsheet file format. In addition to the built-in file formats which are specified by one of the `sfXXXX` declarations, it is possible to provide dedicated reader and writer units to get access to special file formats.

- Write a unit implementing a reader and a writer for the new file format. They must inherit from the basic `TsCustomSpreadReader` and `TsCustomWriter`, respectively, - both are implemented in unit *fpsReaderWriter* -, or from one of the more advanced ones belonging to the built-in file formats.
- Register the new reader/writer by calling the function `RegisterSpreadFileFormat` in the `initialization` section of this unit (implemented in unit *fpsRegFileFormats*):

```
function RegisterSpreadFormat(AFormat: TsSpreadsheetFormat; AReaderClass: TsSpreadReaderClass; AWriterClass: TsSpreadWriterClass;
  AFormatName, ATechnicalName: String; const AFileExtensions: array of String): TsSpreadFormatID;
```

- `AFormat` must have the value `sfUser` to register an external file format.
- `AReaderClass` is the class of the reader (or nil, if reading functionality is not implemented).
- `AWriterClass` is the class of the writer (or nil, if writing functionality is not implemented).
- `AFormatName` defines the name of the format as used for example in the filter list of file-open dialogs.
- `ATechnicalName` defines a shorter format name.
- `AFileExtensions` is an array of file extensions used in the files. The first array element denotes the default extension. The extensions must begin with a period as in `.xls`.
- The registration function returns a numerical value (`TsSpreadFormatID`) which can be used as format identifier in the workbook reading and writing functions which exist in overloaded version accepting a numerical value for the format specifier. In contract to the built-in formats the FormatID is negative.
- Finally, in your application, add the new unit to the `uses` clause. This will call the registrations function when the unit is loaded and make the new file format available to FPSpreadsheet.

Stream selection

Workbooks are loaded and saved by means of the `ReadFromFile` and `WriteToFile` methods, respectively (or by their stream counterparts, `ReadFromStream` and `WriteToStream`).

By default, the data files are accessed by means of **memory streams** which yields the fastest access to the files. In case of very large files (e.g. tens of thousands of rows), however, the system may run out of memory. There are two methods to defer the memory overflow by some extent.

- Add the element `boBufStream` to the workbook's `Options`. In this case, a **"buffered" stream** is used for accessing the data. This kind of stream holds a memory buffer of a given size and swaps data to file if the buffer becomes too small.
- Add the element `boFileStream` to the workbook's `Options`. This option avoids memory streams altogether and creates temporary **files** if needed. This is, however, the slowest method of data access.
- If both options are set then `boBufStream` is ignored.
- In practice, however, the effect of the selected streams is not very large if memory is to be saved.

Virtual mode

Beyond the transient memory usage during reading/writing the main memory consumptions originates in the internal structure of FPSpreadsheet which holds all data in memory. To overcome this limitation, a "virtual mode" has been introduced. In this mode, data are received from a data source (such as a database table) and are passed through to the writer without being collected in the worksheet. It is clear that data loaded in virtual mode cannot be displayed in the visual controls. Virtual mode is good for conversion between different data formats.

These are the steps required to use this mode:

- Activate virtual mode by adding the option `boVirtualMode` to the `Options` of the workbook.
- Tell the spreadsheet writer how many rows and columns are to be written. The corresponding worksheet properties are `VirtualRowCount` and `VirtualColCount`.
- Write an event handler for the event `OnWriteCellData` of the worksheet. This handler gets the index of row and column of the cell currently being saved. You have to return the value which will be saved in this cell. You can also specify a template cell that physically exists in the workbook from which the formatting style is copied to the destination cell. Please be aware that when exporting a database, you are responsible for advancing the dataset pointer to the next database record when writing of a row is complete.
- Call the `WriteToFile` method of the workbook.

Virtual mode also works for reading spreadsheet files.

The folder *example/other* contains a worked out sample project demonstrating virtual mode using random data. More realistic database examples are in *example/db_import_export* and in the chapter on converting a large database table using virtual mode.

Dataset export

FPC contains a set of units that allow you to export datasets to various formats (XML, SQL statements, DBF files,...). There is a master package that allows you to choose an export format at design time or run time (Lazarus package lazdbexport).

FPSpreadsheet has `TFPSExport` which plugs into this system. It allows you to export the contents of a dataset to a new spreadsheet (*.xls*, *.xlsx*, *.ods*, wikitable format) file into a table on the first sheet by default. In addition, if `MultipleSheets` is set to `TRUE` it is possible to combine several sheets into individual worksheets in the same file. You can optionally include the field names as header cells on the first row using the `HeaderRow` properties in the export settings. The export component tries to find the number format of the cells according to the dataset field types.

For more complicated exports, you need to manually code a solution (see examples below) but for simple data transfer/dynamic exports at user request, this unit will probably be sufficient.

A simple example of how this works:

```
uses
...
fpsexport
...
var
  Exp: TFPSExport;
  ExpSettings: TFPSExportFormatSettings;
  TheDate: TDateTime;
begin
  FDataset.First; //assume we have a dataset called FDataset
  Exp := TFPSExport.Create(nil);
  ExpSettings := TFPSExportFormatSettings.Create(true);
  try
    ExpSettings.ExportFormat := efXLS; // choose file format
    ExpSettings.HeaderRow := true; // include header row with field names
    Exp.FormatSettings := ExpSettings; // apply settings to export object
    Exp.Dataset:=FDataset; // specify source
    Exp.FileName := 'c:\temp\datadump.xls';
    Exp.Execute; // run the export
  finally
    Exp.Free;
    ExpSettings.Free;
```

```
  end;
```

Export to DBF

You can save the information from Excel to dbf rather easily. Here is the example from forum. Please note that this example is supposed to work with Win1251 encoding by default. To get more information please refer to this thread [1] (http://forum.lazarus.freepascal.org/index.php/topic,41636.0.html)

```
procedure TForm1.ExportToDBF(AWorksheet: TsWorksheet; AFileName: String);
var
  i: Integer;
  f: TField;
  r, c: Cardinal;
  cell: PCell;
begin
  DbfGlobals.DefaultCreateCodePage := 1251; //default encoding of the dbf
  DbfGlobals.DefaultOpenCodePage := 1251; //default encoding of the dbf
  if Dbf1.Active then Dbf1.Close;
  if FileExists(AFileName) then DeleteFile(AFileName);

  Dbf1.FilePathFull := ExtractFilePath(AFileName);
  Dbf1.TableName := ExtractFileName(AFileName);
  Dbf1.TableLevel := 25;  // DBase IV: 4 - most widely used; or 25 = FoxPro supports nfCurrency
  Dbf1.LanguageID := $C9; //russian language by default
  Dbf1.FieldDefs.Clear;
  //below are the fields in excel
  Dbf1.FieldDefs.Add('fam', ftString);
  //add other fields you want to save

  Dbf1.CreateTable;
  Dbf1.Open;

  for f in Dbf1.Fields do
    f.OnGetText := @DbfGetTextHandler;

  // Skip row 0 which contains the headers
  for r := 1 to AWorksheet.GetLastRowIndex do begin
    Dbf1.Append;
    for c := 0 to Dbf1.FieldDefs.Count-1 do begin
      f := Dbf1.Fields[c];
      cell := AWorksheet.FindCell(r, c);
      if cell = nil then
        f.Value := NULL
      else
        case cell^.ContentType of
          cctUTF8String: f.AsString := UTF8ToCP1251(cell^.UTF8StringValue);
          cctNumber: f.AsFloat := cell^.NumberValue;
          cctDateTime: f.AsDateTime := cell^.DateTimeValue;
          else f.AsString := UTF8ToCP1251(AWorksheet.ReadAsText(cell));
        end;
    end;
    Dbf1.Post;
  end;
end;

//This procedure is called so the text in the cells could be correctly dispayed on WorkSheetGrid on the form, otherwise '?' symbols showing
procedure TForm1.DbfGetTextHandler(Sender: TField; var AText: string; DisplayText: Boolean);
begin
  if DisplayText then
    AText := CP1251ToUTF8(Sender.AsString);
end;
```

Visual controls for FPSpreadsheet

The package **laz_fpspreadsheet_visual** implements a series of controls which simiplify creation of visual GUI applications:

- **TsWorkwookSource** links the controls to a workbook and notifies the controls of changes in the workbook.
- **TsWorksheetGrid** implements a grid control with editing and formatting capabilities; it can be applied in a similar way to TStringGrid.
- **TsWorkbookTabControl** provides a tab sheet for each worksheet of the workbook. It is the ideal container for a TsWorksheetGrid.
- **TsWorksheetIndicator**: a combobox which lists all worksheets of the workbook.
- **TsCellEdit** corresponds to the editing line in Excel or Open/LibreOffice. Direct editing in the grid, however, is possible as well.
- **TsCellIndicator** displays the name of the currently selected cell; it can be used for navigation purposes by entering a cell address string.
- **TsCellCombobox** offers to pick cell properties for a selection of formatting attributes: font name, font size, font color, background color.
- **TsSpreadsheetInspector** is a tool mainly for debugging; it displays various properties of workbook, worksheet, cell value and cell formatting, similar to the ObjectInspector of Lazarus. It is read-only, though.
- Various standard actions are provided in the unit **fpsActions**. Applied to menu or toolbar, they simplify typical formatting and editing tasks without having to write a line of code.
- **TsWorkbookChartLink** and **TsWorkbookChartSource** interface a workbook with the TAChart library. TsWorkbookChartLink reads all chart-related parameters from a workbook chart and display it in a standard Lazarus TAChart. TsWorkbookChartsource defines the cell ranges from which a chart series can get its data.

See the FPSpreadsheet tutorial: Writing a mini spreadsheet application for more information and a tutorial, and see demo projects for examples of the application of these components. Creating charts in a spreadsheet and displaying them by the TAChart library is demonstrated in the chart tutorial.

Examples

Please see FPSpreadsheet: Examples for a variety of code examples.

Download

Subversion

You can download FPSpreadsheet using the subversion software and the following command line:

```
svn checkout https://svn.code.sf.net/p/lazarus-ccr/svn/components/fpspreadsheet fpspreadsheet
```

Stable releases

The most current release is distributed by the Online-Package-Manger which is integrated in the Lazarus IDE (menu "Package" > "Online package manager"): Scroll down the list of available packages, check the entry "FPSpreadsheet" and click "Install". Confirm the prompt to rebuild the IDE. When the process is complete you find the new components in palette "FPSpreadsheet".

Older releases of FPSpreadsheet cann still be found on sourceforge (https://sourceforge.net/projects/lazarus-ccr/files/FPSpreadsheet/).

Installation

- If you only need non-GUI components: in Lazarus: Package/Open Package File, select **laz_fpspreadsheet.lpk**, click Compile. Now the package is known to Lazarus (and should e.g. show up in Package/Package Links). Now you can add a dependency on `laz_fpspreadsheet` in your project options and `fpspreadsheet` to the uses clause of the project units that need to use it.

- If you also want GUI components (TsWorksheetGrid and TsWorksheetChartSource): Package/Open Package File, select **laz_fpspreadsheet_visual.lpk**, click Compile, then click Use, Install and follow the prompts to rebuild Lazarus with the new package. Drop needed grid/chart components on your forms as usual.
- If you want to have a GUI component for dataset export: Package/Open Package File, select **laz_fpspreadsheetexport_visual.lpk**, click Compile, then click, Use, Install and follow the prompts to rebuild Lazarus with the new package. Drop needed export components from the **Data Export** tab on your forms as usual.
- FPSpreadsheet is developed with the latest stable fpc version (currently fpc 3.0.2). We only occasionally check older versions.
- The basic spreadsheet functionality works with Lazarus versions back to version 1.0. Some visual controls or demo programs, however, require newer versions. Please update your Lazarus if you have an older version and experience problems.

Conditional defines

Here is a list of **conditional defines** which can be activated in order to tweak some operating modes of the packages and/or make it compilable with older Lazarus/FPC versions:

- `FPS_DONT_USE_CLOCALE`: In Unix systems, the unit `clocale` is automatically added to the uses clause of *fpspreadsheet.pas*. This unit sets up localization settings needed for locale-dependent number and date/time formats. However, this adds a dependence on the C library to the package [2] (http://lazarus-ccr.sourceforge.net/docs/rtl/clocale/index.html). If this is not wanted, define `FPS_DONT_USE_CLOCALE`.
- `FPS_VARISBOOL`: fpspreadsheet requires the function VarIsBool which was introduced by fpc 2.6.4. If an older FPC versions is used define `FPS_VARISBOOL`. Keep undefined for the current FPC version.
- `FPS_LAZUTF8`: fpspreadsheet requires some functions from the unit LazUTF8 which were introduced by Lazarus 1.2. If an older Lazarus version is used define `FPS_LAZUTF8`. Keep undefined for the current Lazarus version.
- `FPS_NO_GRID_MULTISELECT`: In order to allow selection of multiple ranges in the WorksheetGrid a sufficiently new version of the basic TCustomGrid is needed. The required property "RangeSelect" was introduced in Lazarus 1.4. In order to compile the package with older versions activate the define `FPS_NO_GRID_MULTISELECT`.
- `FPS_PTRINT`: In order to provide safe casting of integers to pointers new version of FPC provide the types PtrInt and IntPtr. This is not yet available in fpc 2.6.0.
- `FPS_NEED_STRINGHASHLIST`: Unit stringhashlist belongs to LCL before Lazarus 1.8. To avoid a requirement of LCL in laz_fpspreadsheet.lpk a copy in the fps directory is provided. This copy is used when the define FPS_NEED_STRINGHASHLIST is active. The define is not needed for Lazarus versions >= 1.8.
- `FPS_NO_NEW_UTF8_ROUTINES`: In Lazarus 2.0+ some UTF8 routines in unit LazUTF8 were renamed from UTF8CharacterXXX to UTF8CodePointXXX. Activate the following define when the new routines are not available, i.e. for Lazarus version < 2.0.
- `FPS_NO_LAZUTF16`: Lazarus 1.8+ has unit LazUTF16 for special access to widestring. This define must be active when this unit is not available, i.e. for Lazarus versions before 1.8.0.
- `FPS_NO_STRING_SPLIT`: Activate the following define if FPS does not have the string Split helper, e.g. before v3.0
- `FPS_CHARTS`: Activate chart support by fpspreadsheet and their reading/writing in xlsx and ods. This feature is behind this define since many spreadsheets do not use charts.

All these defines are collected in the include file *fps.inc*.

In the most recent test (Feb 2023), all fpspreadsheet packages compile fine with Laz 1.6+ / fpc 3.0+ (some defines in *fps.inc* may have to be adapted, though). With Laz 1.4 / fpc 2.6.4 or older, *fpspreadsheet_dataset.lpk* and *fpspreadsheet_crypto.lpk* cannot be used any more.

Support and Bug Reporting

The recommended place to discuss FPSpreadsheet and obtain support is asking in the Lazarus Forum (http://forum.lazarus.freepascal.org/index.php?board=49.0).

Bug reports should be sent to the Lazarus/Free Pascal Bug Tracker (http://bugs.freepascal.org/); please specify the "Lazarus-CCR" project.

Current Progress

Progress by supported cell content

Format	Multiple sheets	Unicode	Reader support	Writer support	Text	Number	String Formula	RPN Formula	3D cell references	Date/ Time	Comments	Hyperlinks	Images + ++	Protection	Conditional formats
CSV files	No	Yes +	Working ++	Working ++	Working ++	Working ++	N/A	N/A	N/A	Working + +	N/A	N/A	N/A	N/A	N/A
Excel 2.x	No	No *	Working **	Working	Working	Working	Working	Working ***	N/A	Working ****	Working	N/A	N/A	Working	N/A
Excel 5.0 (Excel 5.0 and 95)	Yes	No *	Working **	Working	Working	Working	Working	Working ***	Working	Working ****	Working	N/A	N/A	Working	N/A
Excel 8.0 (Excel 97-2003)	Yes	Yes	Working **	Working	Working	Working	Working	Working ***	Working	Working ****	Reading only	Working	Not working	Working	Not working
Excel 2003/XML	Yes	Yes	Working **	Working	Working	Working	Working ***	Working	Working	Working ****	Working	Working	N/A	Working	Working
Excel OOXML	Yes	Yes	Working **	Working	Working	Working	Working ***	Working	Working	Working ****	Working	Working	Working	Working	Working +++ +
OpenDocument	Yes	Yes	Working **	Working	Working	Working	Working ***	Working	Working	Working ****	Working	Working	Working	Working	Working
HTML	No	Yes	Working ++	Working ++	Working ++	Working ++	N/A	N/A	N/A	Working + +	N/A	Working	Not working	N/A	N/A
Wikitable files (Mediawiki)	No	Yes	planned	Working ++	Working ++	Working ++	N/A	N/A	N/A	Working + +	N/A	N/A	Not working	N/A	N/A

(+) Depends on file.
(++) No "true" number format support because the file does not containg number formatting information. But the number format currently used in the spreadsheet understood.
(+++) Only very basic image support: no transformations, no cropping, no image manipulation.
(++++) Extended OOXML formatting options not supported.
(*) In formats which don't support Unicode the data is stored by default as ISO 8859-1 (Latin 1). You can change the encoding in TsWorkbook.Encoding. Note that FPSpreadsheet offers UTF-8 read and write routines, but the data might be converted to ISO when reading or writing to the disk. Be careful that characters which don't fit selected encoding will be lost in those operations. The remarks here are only valid for formats which don't support Unicode.
(**) Some cell could be returned blank due to missing or non ready implemented number and text formats.
(***) This is the format in which the formulas are written to file (determined by design of the file format).
(****) Writing of all formats is supported. Some rare custom formats, however, may not be recognized correctly. BIFF2 supports only built-in formats by design.

Progress of the formatting options

The following formatting options are available:

Format	Text alignment	Text rotation	Font	Rich text	Border	Color support	Back ground	Word wrap	Col&Row size	Number format	Merged cells	Page layout	Print ranges	Header/ footer images	Column/ row format	Hide cols/rows	Page breaks
CSV files	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
Excel 2.x	Working *	N/A	Working	N/A	Working	Working	Working**	N/A	Working	Working	N/A	Working	N/A	N/A	Working	N/A	Working
Excel 5.0 (Excel 5.0 and 95)	Working	Working	Working	Working	Working	Working	Working	Working	Working	Working	N/A	Working	Working	N/A	Working	Working	Working
Excel 8.0 (Excel 97 - XP)	Working	Working	Working	Working	Working	Working	Working	Working	Working	Working	Working	Working	Working	Not working	Working	Working	Working
Excel 2003/XML	Working	Working	Working	Working	Working	Working	Working	Working	Working	Working	Working	Working	Working	N/A	Working	Wworking	Working
Excel OOXML (xlsx)	Working	Working	Working	Working	Working	Working	Working	Working	Working	Working	Working	Working	Working	Working	Working	Working	Working
OpenDocument	Working	Working	Working	Working	Working	Working	Working***	Working	Working	Working	Working	Working	Working	Working	Working	Working	
HTML (+)	Working	bugs	Working	Working	bugs	Working	Working ****	Writing only	Writing only	N/A	Working	N/A	N/A	Not working	Writing only	to be done	N/A
Wikitable (Mediawiki)	Writing only	N/A	Writing only	Not working	Writing only	Writing only		Writing only	Writing only	N/A	Writing only	N/A	N/A	Not working	Writing only	to be done	N/A

(N/A) Feature is not available for this format intrinsically.
(*) BIFF2 supports only horizontal text alignment, vertical alignment is ignored.
(**) BIFF2 does not support a background color; a dotted black&white background is used instead.
(***) OpenDocument supports only uniform backgrounds; a fill color interpolated between foreground and background colors is written instead.
(****) Only uniform background color, no fill styles.
(+) HTML reader does not support styles. Since the writer does use styles these files are not read back correctly.

Progress of workbook/worksheet user-interface and other options

Some additional options were added to interface the file contents with the `TsWorksheetGrid`:

Format	Hide grid lines	Hide headers	Frozen Panes	Active sheet/cell	Zooming	BiDi mode	Tab color	Charts
Excel 2.x	Working	Working	not working	N/A	N/A	N/A	N/A	not working
Excel 5.0 (Excel 5.0 and 95)	Working	Working	Working	Working	Working	N/A	N/A	not working
Excel 8.0 (Excel 97 - XP)	Working	Working	Working	Working	Working	Working	Working	not working
Excel2003/XML	Working	Working	Working	Working	Working	Working	N/A	not working
Excel OOXML	Working	Working	Working	Working	Working	Working	Working	Working
OpenDocument	Working	Working	Working	Working	Working	Working	Working	Working
HTML	Writing only	Writing only	N/A	N/A	N/A	not working	N/A	N/A

To do list

Note: this list is provisional, maintained by developers and can change without notice. If you are interested in contributing, please feel free to get in touch and submit a patch - thanks!

- Find out why BIFF2 files are corrupt when saved with frozen rows/cols activated.
- ~~Add row and column formats~~
- Add reading support for wikitable (Mediawiki) files
- xls reader crashes for some incorrectly written xls files (which Excel can read), see http://forum.lazarus.freepascal.org/index.php/topic,25624.0.html (http://forum.lazarus.freepascal.org/index.php/topic,25624.0.html).
- ~~Improve color support: due to palette support colors may change from file to file currently.~~
- Add embedded images.
- Fix writing of cell comments to BIFF8 files.

Long-term:

- Provide a more common user interface to fpspreadsheet (setter/getter and properties instead of Read*/Write* methods, MyWorksheet.Cells[ARow, ACol]), make TCell a class, avoid the pointers PCell.
- ~~Store formatting in a format list of the workbook, not in the cell, to reduce memory usage.~~
- Use BIFF8 file-wide string storage instead of storing strings in cells (should reduce memory usage in case of many duplicate strings)
- Redo ooxml and ods readers based upon SAX/TXmlReader instead of DOM to reduce memory usage of large files.
- ~~Add an fpspreadsheetexport unit and component on "Data Export" tab similar to fpdbfexport FPSpreadsheetexport demo preferably with all export formats component.~~ Find a way to register export format to all formats (look into how lazreport pdf export performs a similar procedure).

Changelog

SVN

Changes with respect to current release

- (currently none)

Incompatible changes

- (currently none)

Version 2.0.0

This is the latest stable release.

Changes with respect to v1.16

- Supporting charts
- Supporting defined names (named cells and ranges).
- Supporting new formulas: `INDEX, ROUNDDOWN, ROUNDUP, COUNTIF, SUMF, AVERAGEIF, COUNTIFS, SUMIFS, AVERAGEIFS`
- Supporting horizontal alignments `haDistributed` and `haFilled`

Incompatible changes

- (currently none)

Version 1.16

Changes with respect to v1.14

- XLSX reader can handle embedded as well as header/footer images now.
- Less strict format check by Excel2 reader since there are BIFF2 files with BOF record of Excel8 which Excel can read.
- Reading support for the Excel3 and Excel4 formats (unit *xlsbiff34*)
- ODS reader supports the <text:s> node
- ODS reader is able to open encrypted files.

Incompatible changes

- (none)

Version 1.14

Changes with respect to v1.12

- Implementation of **conditional formats** for ods, xlsx and Excel xml.
- Implementation of **meta data** for ods, xlsx and Excel xml.
- Add a worksheet-based dataset in separate package (**TsWorksheetDataset**).

Incompatible changes

- Removed the deprecated worksheet properties DefaultRowHeight and DefaultColWidth. Use the corresponding Read/Write routines instead (ReadDefaultColWidth(units), WriteDefaultColWidth(units)).

Version 1.12

Changes with respect to v1.10x

- **Hide** (and unhide) rows and columns.
- Enforce **Page breaks** before rows and columns during printing by the Office applications.
- Full support (reading and writing) of the **ExcelXML** xml format (Excel 2003 and XP).
- Support of the color of worksheet tabs (Worksheet.TabColor) in xls biff8, xlsx and ods formats. The property is ignored by the visual WorkbookTabControl, though.
- Add TsWorksheetIndicator to visual controls.

Version 1.10.1

Changes with respect to v1.8x

- **Workbook**, **worksheet and cell protection** (read/write in BIFF2/BIFF5/BIFF8/OOXML/ODS, write in ExcelXML).
- New package **laz_fpspreadsheet_crypto** to decipher the encryption for worksheet-protection in xls files. Requires DCPcrypt.
- TsWorksheetGrid can **display embedded images**.
- **Drag and drop** in TsWorksheetGrid
- New, **highDPI-aware** component palette icons.
- New workbook optiond **boAbortReadingOnFormulaError** and **boIgnoreFormulas**.
- Formulas with **references to other sheets**, i.e. `'=Sheet1!A1+Sheet2!A2'`

Incompatible changes

- The field **FormulaValue** has been removed from the cell record. The cell formula now can be retrieved by calling `Worksheet.ReadFormula(cell)`.

Version 1.8

Changes with respect to v1.6x

- **"Rich-text" formatting** of label cells, i.e. assignment different fonts to groups of characters within the cell text. For this purpose, HTML codes (such as `<B>...</B>`) can be embedded in the cell text to identify the parts with different font (--> `TsWorksheet.WriteTextAsHTML`).
- **Searching** for cells with specified content in worksheet or workbook.
- Support for reading and writing of **HTML format**
- Support for writing of the **ExcelXML format** (Excel XP and 2003)
- Ability to use **user-provided file reader/writer classes** to extend FPSpreadsheet to new file formats.
- Readers and writers now support all the **line styles** of Excel8 and OOXML.
- xls, xlsx and ods readers/writers now support the **active worksheet** and **selected cell**.
- Ability to write to/read from the system's **clipboard** for copy & paste using the visual controls.
- Support for **print ranges** and **repeated header rows and columns** in the Office applications.
- Support for **embedded images** (currently only writing to xlsx and ods, no reading).
- Improved compatibility of **TsWorksheetGrid** with TStringGrid (`Cells[Col,Row]` property). Standalone application as an advanced StringGrid replacement.
- Support for **Right-to-left mode** in TsWorksheetGrid. In addition to the system-wide RTL mode, there are also parameters `BiDiMode` in the Worksheet and cells allowing to controls text direction at worksheet and cell level individually, like in Excel or LibreOffice Calc.
- Support of several **units** for specification of column widths and row heights.
- The library now supports **localization** using po files. Translations are welcome.
- **Zoom factor** read and written by the worksheet, and applied by the TsWorksheetGrid.
- Support of **column and row formats**
- Support of **hidden** worksheets

Incompatible changes

- VirtualMode was changed in order to be able to treat worksheets of the same workbook differently. `VirtualRowCount` and `VirtualColCount` are now properties of the worksheet, and similarly, the event handler `OnWriteCellData`. In older versions, these properties had belonged to the workbook.
- The worksheet methods `ReadAsUTF8Text` and `WriteUTF8Text` have been renamed to `ReadAsText` and `WriteText`, respectively. The old ones are still available and marked as deprecated; they will be removed in later versions.
- The public properties of `TsWorksheetGrid` using a `TRect` as parameter were modified to use the Left, Top, Right, Bottom values separately.
- The `PageLayout` is a class now, no longer a record. As a consequence, some array properties cannot be set directly any more, use the corresponding methods instead.
- Most of the predefined color constants were marked as deprecated; only the basic EGA colors will remain.

- Unit fpsNumFormatParser is integrated in fpsNumFormat. Old code which "uses" fpsNumFormatParser must "use" fpsNumFormat now.
- The source files of the laz_fpspreadsheet, laz_fpspreadsheet_visual and laz_fpspreadsheetexport_visual packages have been moved to separate folders in order to resolve some occasional compilation issues. Projects which do not use the packages but the path to the sources must adapt the paths.

Version 1.6

Changes with respect to v1.4.x

- TsWorkbookChartSource is a new component which facilitates creation of charts from non-contiguous spreadsheet data in various worksheets. It interfaces to a workbook via tha WorkbookSource component. In the long run, it will replace the older TsWorksheetChartSource which required contiguous x/y data blocks in the same worksheet.
- Major reconstruction of the cell record resulting in strong reduction of memory consumption per cell (from about 160 bytes per cell down to about 50)
- Implementation of a record helper for the TCell which simplifies cell formatting (no need to set a bit in UsedFormattingFields any more, automatic notification of visual controls)
- Comments in cells
- Background fill patterns
- Hyperlinks
- Enumerators for worksheet's internal AVLTrees for faster iteration using a for-in loop.
- Formatting of numbers as fractions.
- Improved number format parser for better recognition of Excel-like number formats.
- Page layout (page margins, headers, footer, used for only when printing in the Office applications - no direct print support in fpspreadsheet!)
- Improved color management: no more palettes, but direct rgb colors. More pre-defined colors.
- A snapshot of the wiki documentation is added to the library as chm help file.

Incompatible changes

- All type declarations and constants are moved from fpspreadsheet.pas to the new unit **fpstypes.pas**. Therefore, most probably, this unit has to be added to the uses clause.
- Because fpspreadsheet supports now **background** fill patterns the cell property `BackgroundColor` has been replaced by `Background`. Similarly, the `UsedFormattingFields` flag `uffBackgroundColor` is called `uffBackground` now.
- Another `UsedFormattingFields` flag has been dropped: **`uffBold`**. It is from the early days of fpspreadsheet and has become obsolete since the introduction of full font support. For achieving a bold type-face, now call `MyWorksheet.WriteFont(row, col, BOLD_FONTINDEX)`, or `Myworksheet.WriteFontStyle(row, col, [fssBold])`.
- **Iteration through cells** using the worksheet methods `GetFirstCell` and `GetNextCell` has been removed - it failed if another iteration of this kind was called within the loop. Use the new `for-in` syntax instead.
- Support for **shared formulas** has been reduced. The field `SharedFormulaBase` has been deleted from the TCell record, and methods related to shared formulas have been removed from TsWorksheet. Files containing shared formulas can still be read, the shared formulas are converted to multiple normal formulas.
- The **color palettes** of previous versions have been abandoned. `TsColor` is now a `DWord` representing the rgb components of a color (just like TColor does in the *graphics* unit), it is not an index into a color palette any more. The values of pre-defined colors, therefore, have changed, their names, however, are still existing. The workbook functions for palette access have become obsolete and were removed.

Version 1.4

Changes with respect to v1.2.x

- Full support for **string formulas**; calculation of RPN and string formulas for all built-in formulas either directly or by means of registration mechanism. Calculation occurs when a workbook is saved (activate workbook option `boCalcBeforeSaving`) or when cell content changes (active workbook option `boAutoCalc`).
- **Shared formulas** (reading for sfExcel5, sfExcel8, sfOOXML; writing for sfExcel2, sfExcel5, sfExcel8).
- Significant **speed-up** of writing of large spreadsheets for the xml-based formats (ods and xlsx), speed up for biff2; **speedtest demo** program
- **VirtualMode** allowing to read and write very large spreadsheet files without loading entire document representation into memory. Formatting of cells in VirtualMode.
- Demo program for database export using virtual mode and TFPSExport.
- Added db export unit allowing programmatic exporting datasets using **TFPSExport**. Similar export units are e.g. fpdbfexport, fpXMLXSDExport.
- Reader for **xlsx** files, now fully supporting the same features as the other readers.
- Reader/writer for **CSV files** based on CsvDocument.
- **Wikitables writer** supports now most of the fpspreadsheet formatting options (background color, font style, font color, text alignment, cell borders/line styles/line colors, merged cells, column widths, row heights); new "wikitablemaker" demo
- **Insertion and deletion of rows and columns** into a worksheet containing data.
- Implementation of **sorting** of a worksheet.
- Support of **diagonal "border" lines**
- **Logging of non-fatal error messages** during reading/writing (TsWorksheet.ErrorMsg)
- **Merged** cells
- **Registration** of currency strings for automatic conversion of strings to **currency values**
- A set of **visual controls** (TsWorkbookSource, TsWorkbookTabControl, TsSpreadsheetInspector, TsCellEdit, TsCellIndicator, TsCellCombobox, in addition to the already-existing TsWorksheetGrid) and pre-defined standard actions to facilitate creation of GUI applications.
- **Overflow cells** in TsWorksheetGrid: label cells with text longer than the cell width extend into the neighboring cell(s).

Incompatible changes

- The option `soCalcBeforeSaving` now belongs to the workbook, no longer to the worksheet, and has been renamed to `boCalcBeforeSaving` (it controls automatic calculation of formulas when a workbook is saved).
- The workbook property `ReadFormulas` is replaced by the option flag `boReadFormulas`. This means that you have to add this flag to the workbook's `Options` in order to activate reading of formulas.
- With full support of string formulas some features related to RPN formulas were removed:
  - The field `RPNFormulaResult` of TCell was dropped, as well as the element `cctRPNFormula` in the `TsContentType` set.
  - Sheet function identifiers were removed from the `TsFormulaElement` set, which was truncated after `fekParen`.
  - To identify a sheet function, its name must be passed to the function `RPNFunc` (instead of using the now removed `fekXXXX` token). In the array notation of the RPN formula, a sheet function is identified by the new token `fekFunc`.
  - The calling convention for registering user-defined functions was modified. It now also requires the Excel ID of the function (see "OpenOffice Documentation of Microsoft Excel Files", section 3.11, or unit *xlsconst* containing all token up to ID 200 and some above).
  - Code related to RPN formulas was moved to a separate unit, `fpsRPN`. Add this unit to the uses clause if you need RPN features.

License

LGPL with static linking exception. This is the same license as is used in the Lazarus Component Library.

References

Documentation

FPSpreadsheet is documented in this wiki file.

Every release of FPSpreadsheet is accompanied by a snapshot of the current wiki files. There is also an api documentation created from the embedded comments in the source files. Both files are in the chm format.

This wiki page is work in progress and updated whenever a new feature is added; therefore, its state corresponds to the svn trunk version of the package. If you work with an older stable version please use these "historic" wiki versions:

- Version 1.16: wiki.lazarus.freepascal.org/index.php?title=FPSpreadsheet&oldid=157438 (https://wiki.lazarus.freepascal.org/index.php?title=FPSpreadsheet&oldid=157438)
- Version 1.14: wiki.lazarus.freepascal.org/index.php?title=FPSpreadsheet&oldid=149828 (https://wiki.lazarus.freepascal.org/index.php?title=FPSpreadsheet&oldid=149828)
- Version 1.12: wiki.lazarus.freepascal.org/index.php?title=FPSpreadsheet&oldid=136886 (https://wiki.lazarus.freepascal.org/index.php?title=FPSpreadsheet&oldid=118771)
- Version 1.10.1: wiki.lazarus.freepascal.org/index.php?title=FPSpreadsheet&oldid=118771 (https://wiki.lazarus.freepascal.org/index.php?title=FPSpreadsheet&oldid=118771)
- Version 1.10: wiki.lazarus.freepascal.org/index.php?title=FPSpreadsheet&oldid=118441 (https://wiki.lazarus.freepascal.org/index.php?title=FPSpreadsheet&oldid=118441)
- Version 1.8: wiki.lazarus.freepascal.org/index.php?title=FPSpreadsheet&oldid=107616 (https://wiki.lazarus.freepascal.org/index.php?title=FPSpreadsheet&oldid=107616)
- Version 1.6.2: wiki.lazarus.freepascal.org/index.php?title=FPSpreadsheet&oldid=100723 (https://wiki.lazarus.freepascal.org/index.php?title=FPSpreadsheet&oldid=100723)
- Version 1.6: wiki.lazarus.freepascal.org/index.php?title=FPSpreadsheet&oldid=91469 (https://wiki.lazarus.freepascal.org/index.php?title=FPSpreadsheet&oldid=91469)
- Version 1.4: wiki.lazarus.freepascal.org/index.php?title=FPSpreadsheet&oldid=85299 (https://wiki.lazarus.freepascal.org/index.php?title=FPSpreadsheet&oldid=85299)
- Version 1.2 and 1.2.1: wiki.lazarus.freepascal.org/index.php?title=FPSpreadsheet&oldid=81375 (https://wiki.lazarus.freepascal.org/index.php?title=FPSpreadsheet&oldid=81375)

Wiki links

- Office Automation
- CsvDocument
- FPSpreadsheet: Examples
- RPN formulas in FPSpreadsheet
- FPSpreadsheet: List of formulas
- FPSpreadsheet tutorial: Writing a mini spreadsheet application
- FPSpreadsheet: Chart Tutorial

External Links

- Microsoft OLE Document Format (http://sc.openoffice.org/compdocfileformat.pdf)
- Excel file format description (http://sc.openoffice.org/excelfileformat.pdf)
- Excel xls and PowerPoint ppt file dumper written in Python - very handy to list all contents of BIFF files (e.g. ./xls-dump.py file.xls) - http://cgit.freedesktop.org/libreoffice/contrib/mso-dumper/ (http://cgit.freedesktop.org/libreoffice/contrib/mso-dumper/). A similar application is the "BIFFExplorer" which can be found in the *applications* folder of ccr.
- Icons used in the demo programs:
  - silk icons (http://www.famfamfam.com/lab/icons/silk/)
  - fugue icons (http://p.yusukekamiyamane.com/)

Retrieved from "https://wiki.freepascal.org/index.php?title=FPSpreadsheet&oldid=162931"
