Flexcel Reports Designers Guide

Documentation: May, 2010

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

Table of contents

Introduction

This document has two different target audiences, developers using FlexCel and final power users that want to customize their reports. It covers how to design an Excel template, but not the code parts needed to run a report. Read the Reports Developer guide for the code part.

Report Elements

There are three concepts you should understand to create or modify a report.

Tags, Named ranges and the **Configuration sheet**.

Tags

A tag is text that you write in a cell and that will be replaced by a different value on the generated report. All tags are on the form **<#TagName>** when they don't have any parameters, and **<#TagName(param1;param2...)>** when they have parameters. Notice that the parameter separator is ";" not "," as it is on expressions.

Tags are case insensitive, so you can write **<#tag>, <#TAG>** or **<#Tag>** as you prefer. The convention we usually use is all lowercase, but it is up to you.

You can write multiple tags on the same place, and the result will be the concatenated string. You may also apply different format to different tags. For example, writing "<#value1> and <#value2>" inside a cell will be replaced by something similar to "1 and 2"

Tags will be replaced on Cells, Comments, Sheet names, Images, Hyperlinks, AutoShapes, Headers and Footers.

Tag Reference:
The complete list of tags you can use and their descriptions is on the file FlexCelReportTags.xls.

Evaluating Expressions
Expressions can be used inside <#If> and <#Evaluate> tags. They behave like standard Excel formulas, and you can use any formula that FlexCel can calculate. But, different from formulas, you can also enter tags inside expressions.

For example, you could write: <#Evaluate(A1+Min(A2,<#Value>))>

Note that the parameter separator on Expressions is "," , not ";" as it is on tags. This is to keep it syntactically compatible with Excel.

The supported list of things you can write inside an expression is detailed on the following table:

Expression	
Tags	**Syntax:** <#Tag>
	Description: You can enter any tag inside an expression, and it will be evaluated. The tag might contain nested expressions too.
	Example: 1+<#Value> will return the report variable "Value" plus 1.
References	**Syntax:** A1, $A1, Sheet1!A2, A1:A2, Sheet1:Sheet2!A1:B20, etc
	Description: Standard Excel cell references. You can use relative and absolute references too.
	Example:

	A1+A2 will return the sum of what is on cell A1 and A2. As the references are not absolute, when copied down this expression will refer to A3, A4, etc.
Parenthesis	**Syntax:** ()
	Description: Changes operator precedence. Standard operator precedence on expressions is the same as in Excel, that is "1+2*3" = 1+(2*3)=7 and not (1+2)*3=9
	Example: (1+2)*3^2 will be evaluated different than 1+2*3^2
Arithmetic Operators	**Syntax:** +, -, *, /, %, ^ (power)
	Description: Standard arithmetic operators.
	Example: 1+2*3^2 will evaluate to 19 5% will evaluate to 0.05
Equality Operators	**Syntax:** <, >, =, >=, <=, <>
	Description: Standard equality operators.
	Example: 1>=2 will evaluate to false.
Functions	You can use any formula function that FlexCel can recalculate inside an expression. For a list of supported functions, take a look at SupportedFunctions.xls

Named Ranges

While Tags allow you to replace complex expressions inside a sheet, with them alone we can only make "Fill in the blanks" type reports. That is, reports that are static, like a form, and where cells with tags will be replaced with their corresponding values.

Now we are going to introduce the concept of "Band" A Band is just a range of cells that is repeated for each record of a table. Imagine that you define a Band on the cell range A1:C1, and associate it with the table "Customer". Then, on cells A1:C1 you will have the first customer, on cells A2:C2 the second and so on. All cells that were previously on A2:C2 will be moved down after the last record of the dataset.

If table customer has six registers, you will get something like:

Template:

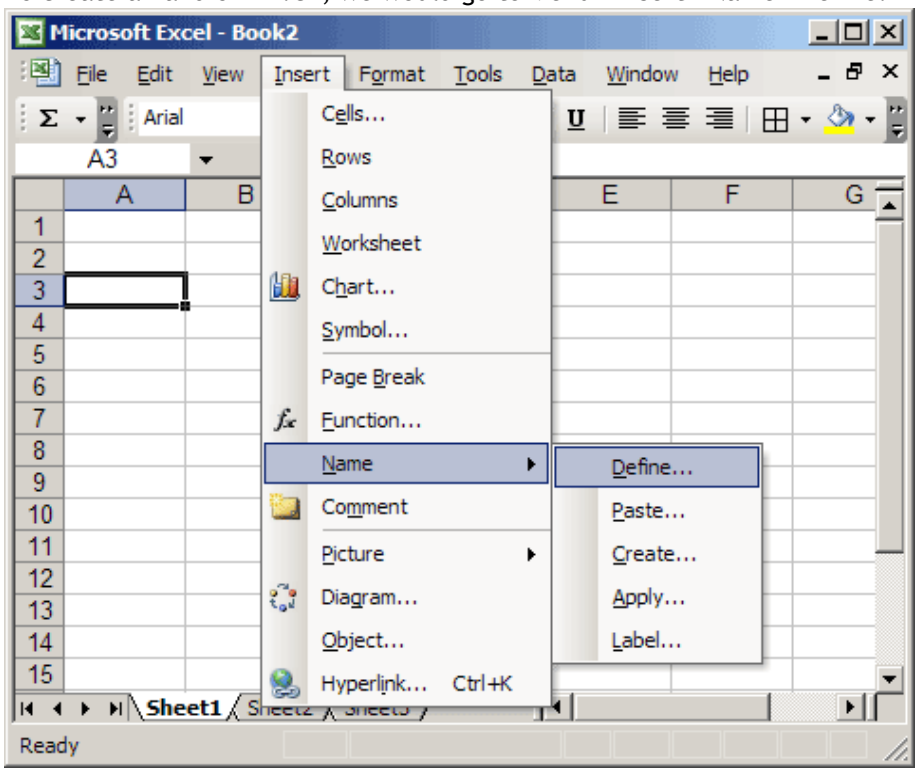
	A	B	C	D
1	<#Customer.CustomerName>	<#Customer.Id>	<#Customer.Age>	
2	This is some text below the band.			
3				
4				

Generated report:

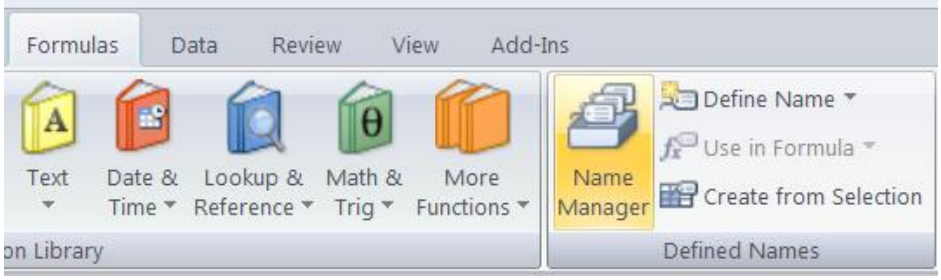
	A	B	C	D
1	Customer1	1	32	
2	Customer2	2	33	
3	Customer3	3	12	
4	Customer4	4	34	
5	Customer5	5	44	
6	Customer6	6	98	
7	This is some text below the band.			
8				
9				

On FlexCel we use Named Ranges to indicate the bands. If you are not used to Excel Named Ranges, take some time to familiarize with them, as they are one of the things that confuses most people starting with FlexCel. Different from tags, that you can immediately see when you open a workbook, named ranges are a little more hidden.

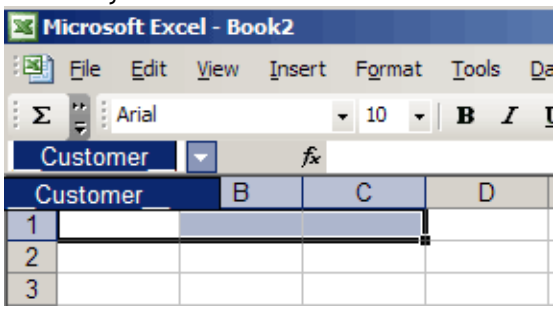
To create a Band on A1:C1, we would go to Menu->Insert->Name->Define.



Or in Excel 2007, go to the "Formulas" tab, then choose "Name Manager":



Once there, we can define a Band __Customer__ on cells A1:C1. And once the name is defined, we can easily see it on the Names combo:



Note the "__" at the beginning and at the end of the range name. We use this to indicate FlexCel that this is a horizontal range that inserts full rows down.

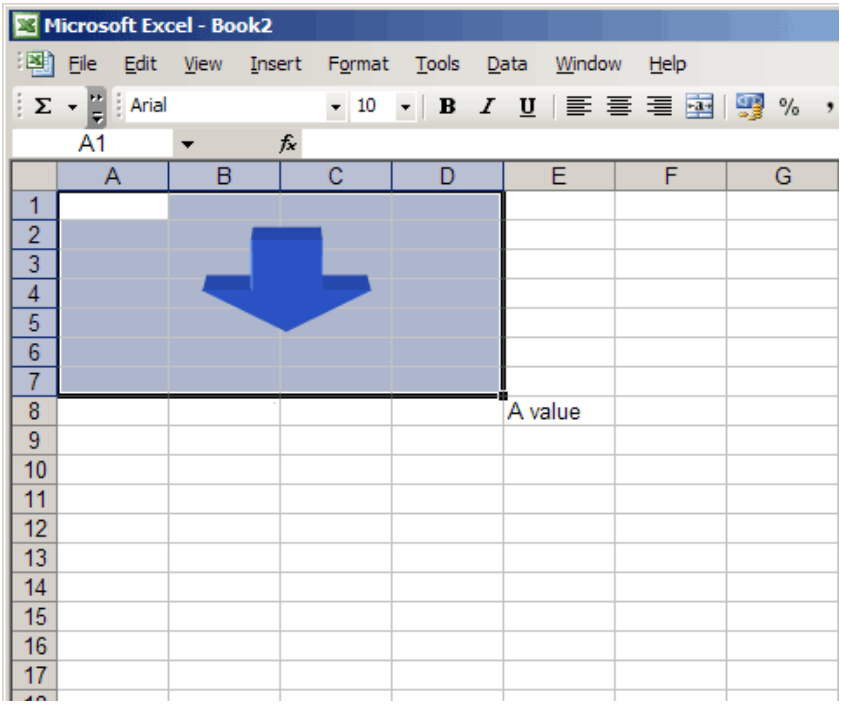
The rest of the name ("Customer") should be the name of an existing datatable, or a custom table defined on the config sheet.

Range Types
You can define four different kinds of bands on FlexCel:

- "__" Range: This range moves down and inserts full rows.
- "_" Range: This range is similar to "__" but cells outside of the range won't move down.
- "II_" Range: This range moves to the right and inserts full columns.
- "I_" Range: This range is similar to "II_" but cells outside of the range won't move right.

A "__" range is the same as a "_" range defined on the full rows, and the same is valid for "II_" and "I_" ranges.

On the following example, if you name A1:D7 as "_Customer_" Cell E8 won't move when inserting down. If you name it as "__Customer__" Cell E8 will move to the last inserted cell, because a "__Customer__" range is equivalent to a "_Customer_" range on A1:IV7.



Master detail
Named ranges can be placed inside others, and a master detail relationship will be automatically created. For example, If you define a range "__Customer__" and inside it a range "__Orders__" and there is a relationship created on the application between "Customer" and "Order" tables, it will automatically group your orders by customer.

On the following example, the yellow cells are the range "__Customer__" and the blue ones are the range "__Orders__"

	A	B	C	D	E	F
1						
2	Customer:	<#Customer.Name>				
3		Orders				
4		<Order.Id>				
5						
6						

After running this report, you will get something similar to:

	A	B	C	D	E	F
1						
2	Customer:	Customer1				
3		Orders				
4		Order1 for Customer1				
5		Order2 for Customer1				
6		Order3 for Customer1				
7	Customer:	Customer2				
8		Orders				
9		Order1 for Customer2				
10		Order2 for Customer2				
11	Customer:	Customer3				
12		Orders				
13		Order1 for Customer3				
14		Order2 for Customer3				
15		Order3 for Customer3				
16						
17						

As you can see, Orders are filtered for each customer, based on the **Data Relationship** defined on the application, and on the nesting on the ranges. In general, any range that is inside another is filtered by all of its parents. You can have as many levels of master-detail relationships as you wish, and each master band filters all of its children.

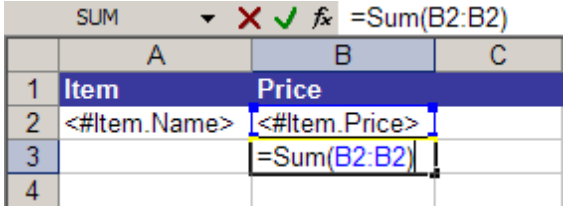
For example, if we wanted to group the customers by country we could define a __Countries__ named range on A1:F6, and it would automatically filter the data on its child and grand-child. (Customer and Order)

There is a special Table that if present filters all the others on the sheet, acting as a parent of all the named ranges on the sheet. This is the table that you define on the name of the sheet, when doing a multiple sheet report. You can see the multiple sheet report demo for more information, there every table on each sheet is filtered by category.

"X" ranges
One issue that might appear when defining named ranges is how formulas on other ranges change when inserting the new cells.

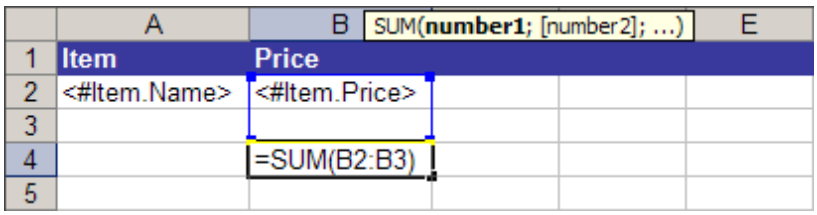
Let's imagine we want to make a simple report on a list of items and their prices.

So, we create a new template, define a title row, and insert a named range on A2:B2 to propagate the data. But we also want to know all the total cost of all items, so we add a formula on B3:



When you run this report, rows will be inserted between row 2 and 3, but the formula Sum(B2:B2) won't change. Nothing has been inserted between B2 and B2, so the sum range will remain constant.

So, we need to have a Sum range that can expand. We will define:



Now, when rows are inserted between row 2 and 3, the formula will be updated to reflect the new rows.

On this particular case, this solution might be enough. Just leave an empty row after the range so formulas expand, and then you can hide the extra row or just leave it there.

But, if we were for example creating a chart, this extra row will be on the chart too. If you are an old FlexCel VCL user, you know about the "...delete row..." thing just to avoid those cases.

Well, delete row does not work anymore on FlexCel 3.0 and up, because the row would be deleted before the range is expanded, and the formula will then point to B2:B2 again.

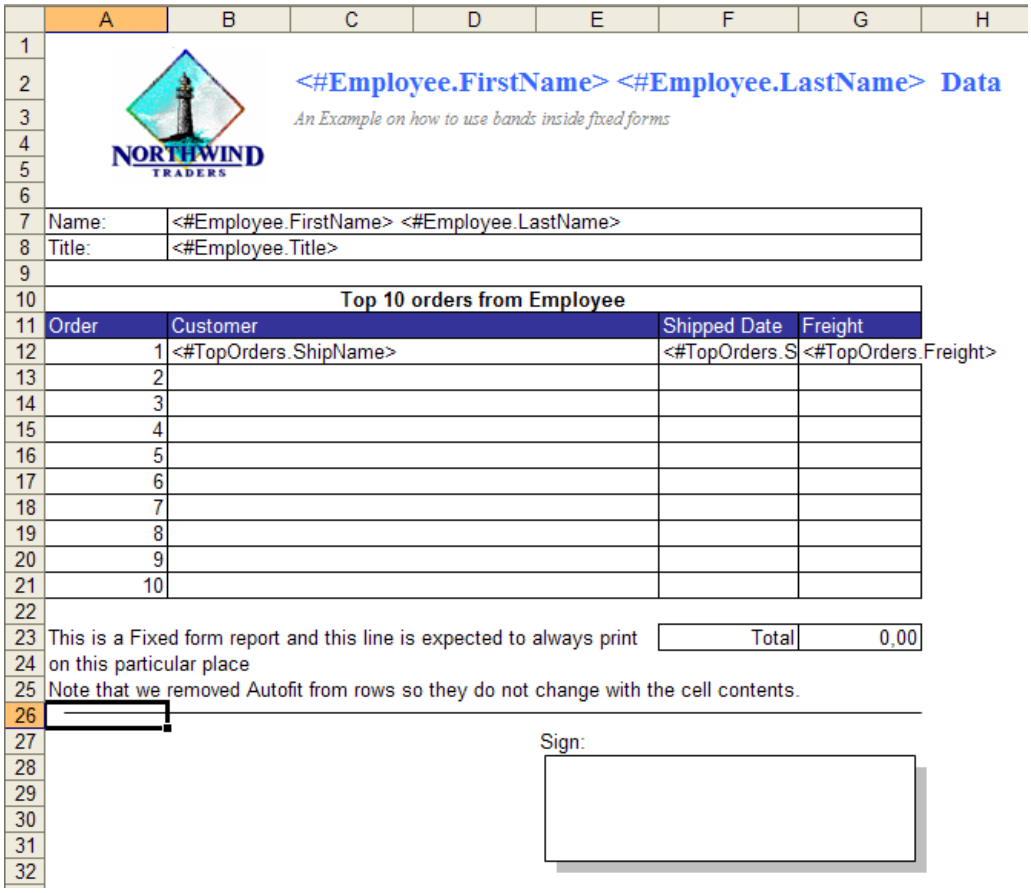
This is why we introduced the "**X" ranges**. X ranges are normal named ranges with an "X" at the end. On this case, instead of "__Item__", we would call the range "__Item__X" It will behave exactly the same as a normal range, but once it is expanded it will erase the last row (or column if it is a column range). So if we try the last example with "__Item__X", row 5 on the last screenshot will be deleted, and the formula would be "=SUM(B2:B4)". Just what we were looking for. See the chart demo for more information on using X ranges.

Fixed Bands
By default, FlexCel will always insert cells when expanding ranges, and this is what you would normally want. If you have a template:

```
A1:Title
A2:<#data>
A3:Footer
```

You would expect that the generated report will have the Footer for example on cell A33 (if we had 30 data records), but not on A3.

But there is a situation where this is not what you expect, and this is on Fixed Form reports. Let's imagine that you want to fill out a form with FlexCel. Most fields will be just simple expressions, not related to datasets, but we might have a table too:



Here, no matter if the dataset has 1 record, two or 10 (it should not have more than 10) you want the "Total" line to be at row 23. You cannot do this with normal ranges, since you would be inserting rows. For this you can define a "__TopOrders__FIXED" named range, that will not insert any records. See the "Fixed Forms with Datasets" demo for more information.

Intelligent Page Breaks
Other kind of ranges you might want to create are "KeepTogether" ranges.

FlexCel will try to keep rows or columns on those ranges together when printing by inserting page breaks at the needed places.

To create a "Row" KeepTogether range, you need to name it:

```
KEEPROWS_<Level>_<Whatever>
```

Where <Level> is the level of "keep together" of the group, and "Whatever" is anything, you can use it to have more than one "keeptogether" range in the same sheet.

For example, you might have the ranges:

```
KeepRows_1_customers
```

and

```
KeepRows_1_orders
```

in the same sheet, to tell FlexCel to group the rows in both ranges together when printing. You might also have different levels of "Keep together", and you will normally use higher levels for details in master-detail reports.

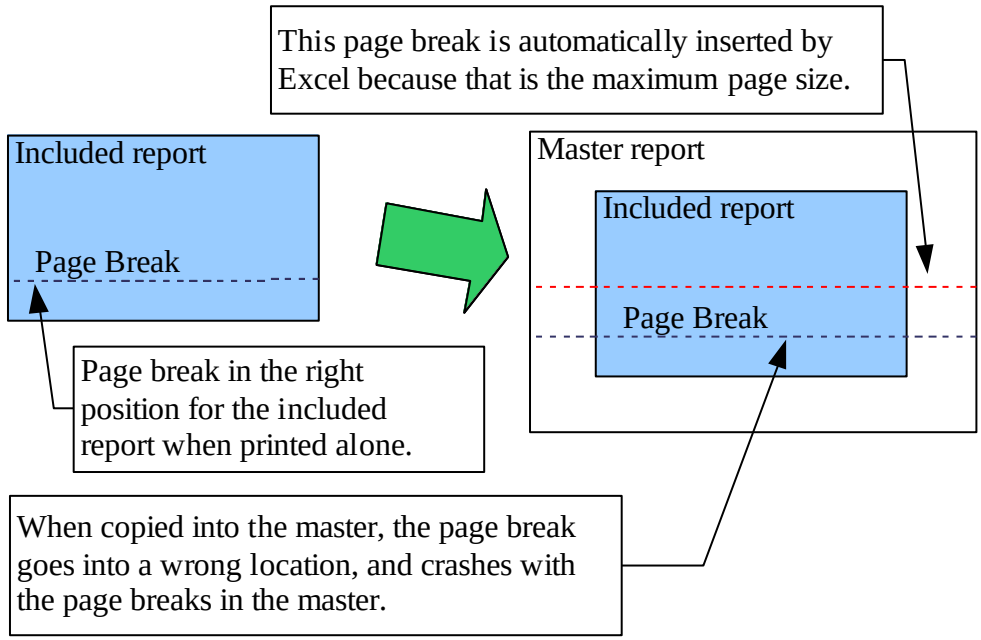
Once you have created the ranges, you need to write an <#auto page breaks> tag somewhere in the sheet, and FlexCel will add page breaks when it ends the report trying to keep those ranges together. You can customize the <#auto page breaks> to influence the way page breaks are created.

We are not covering in detail Intelligent page breaks here, since they are described in the API guide, and it makes no sense to repeat that information here. We only cover here what is different for reports, and this is the "KeepTogether" ranges.

So please make sure you read the section **Preparing for Printing** in the API guide, since most of the concepts apply also to reports. And of course take a look at the **Intelligent Page Breaks** in **Reports** demo.

Intelligent Page Breaks in Included Reports
You can use intelligent page breaks inside an <#included> report, but you must be aware of some differences.

The most important difference is that the final pagination must be done in the master report, not in the included one. Let's look at an example:



In this example, we can have "KeepRows_" and "KeepColumns_" ranges in the included report, but we should not create the actual page breaks in it. If we did it, those page breaks would be in the wrong position when inserted in the master. (unless the report is included at cell A1). As you can see in the drawing, the included report is copied into the master at a lower location. So if the report goes down, the page breaks will go down too, ending up at the wrong places. Excel needs to insert its own page break (marked with a red dotted line) in order to have the master page no bigger than the paper size, and the old page break will create a small page that we don't want.

So, in order to have intelligent page breaks in included report, you must follow the next simple rules:

1. Create the “KeepRows_” and “KeepColumns_” ranges in the included report, as you would normally do it.

2. **Do not write an <#auto page breaks> tag in the included report**. This will ensure FlexCel does not add the page breaks in the included report before copying it to the master.

3. Make sure you include full rows (by using “__” or “II_” as parameter in the include tag). This way, the KeepRows and KeepColumns ranges will be copied to the master.

4. Write an <auto page breaks> tag into the master. This way, when the master report is finished, FlexCel will paginate the master, keeping together the rows you marked in the included report. **Pagination must be done on the master, never in the included report.**

Configuration Sheet

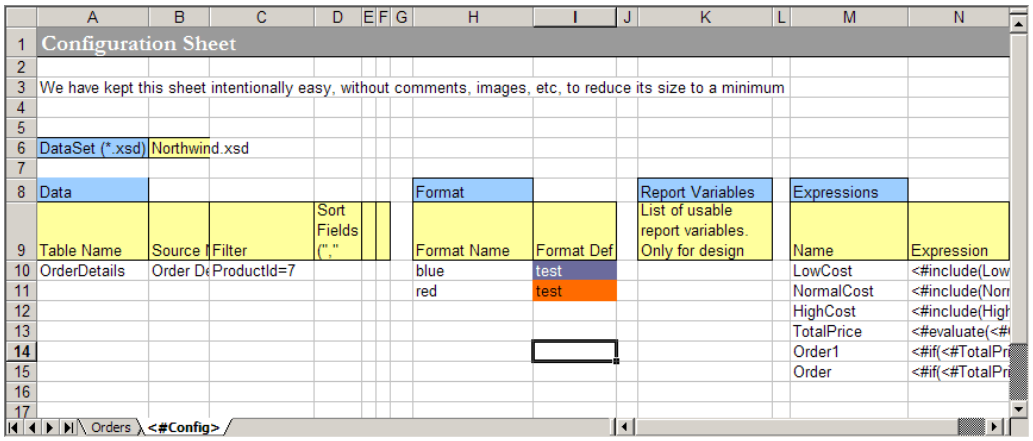
The configuration sheet is a repository where you can configure different things on your report, define common used expressions and so on. It is not required, you can create a report without a configuration sheet, but it is recommended that you have one except for very simple reports.

Once the report is run, the configuration sheet will be deleted.

Note: If there are any macros assigned to the configuration sheet, FlexCel will be unable to delete it and will just clear and “Very hide” the sheet. (Very hidden sheets are similar to hidden sheets but the user needs to write a macro to see them, they are not listed when you select to unhide a sheet from Excel). If you get a macro by mistake on the config sheet and you are unable to delete it in Excel, use the “MacroCleaner” tool included on the “Tools” folder.

The layout of the config sheet is free as long as you keep the positions, you can write the captions you want or change any format you need. Just one advice: keep it simple. The config sheet will be deleted from the final report, so the final user won't see it. And deleting a sheet with comments, images formulas and so on is more time consuming than deleting a simple sheet.

An example of a config sheet is shown here:



We can note that:

1. For a sheet to be the configuration sheet it must be named “<#Config>” or any expression that evaluates to the string “<#Config>”. You could conditionally make a sheet the configuration sheet depending on some parameter.

2. On cell B6 we need to write the name of the schema definition file for the dataset. This is not required, but if you don't do it, FlexCel designer won't be able to show the available tables.

3. Cells A10 to C10 and all rows below (A11:C11... etc) are used to define new custom tables. You use a master table defined by the application (B10), and define a new name for it (A10). Then you can write a filter (any filter expression available on a DataView is ok here, for more info search at the DataView.Filter documentation on the .NET framework docs). And also sort the table by some columns. Different search columns are separated by “,”, and you an also define an “ASC(ending)” or “DESC(ending)” order. For example: “State, ZipCode DESC” Again, any Sort expression valid for a DataView is valid here. For more information consult the .NET framework docs on DataView.Sort.

4. On some places you can use tags. For example, if you write on C10 “<#mytag>” instead of “ProductId=7” the value of mytag wil be used as a filter.

5. You can leave empty rows. For example, you can define a new table on row 10 and another on row 12 without writing anything on row 11. This allows you to better separate your tables.

6. On columns H and I you define custom formats. The name of the format goes on column H, and the definition on column I. Then you can use those cell formats on <#format> tags.

7. You can write whatever you want on column I, the text is not important and you can use it to know how the format will look like.

8. Column K is used to list the report variables. It is not used at all by FlexCel and you don't really need to fill it, but it is used by FlexCel designer to list the available report variables. If you do not write them here, FlexCel designer will not show them.

9. Columns M and N are probably the most important on the config sheet and are used to create reusable expressions.

About Expressions

It is recommended that whenever you have complex chains of tags, instead of writing them directly on the cell, you create an expression and then refer to it.

On the screenshot, we defined a <#order> tag that itself calls other report expressions to calculate its value. Then, if you want to write the result of order on cell A1, instead of writing the full chain of tags, you just write <#Order> On some places, like for example the name of the sheet or an image name this might be the only choice, as the name length is limited to 32 characters. You can use also parameters on Expressions, and for Example define <#order(row)> For more details on how to do this, see the Expressions with parameters demo.

Defining Custom Formats

Custom formats are normally straightforward to use.

For example, you might format I10 with blue background, name it “Header” in cell H10, and then use the format inside a <#format cell(Header)> in the template. But this will apply the full format in cell H10 to the new cell. This means that besides the blue background, all other properties will be copied too. The new cell will also have the same font, same alignment, same numeric format than H10.

Now let's imagine we have a full row that we want to be blue if some condition applies. If we write <#format row(header)> in a cell on the template, then all other attributes besides the background will be applied to the row. This might mean that all the cells will be aligned to the left (if cell I10

was aligned to the left), and this is not what we want. We want to apply only the background color on I10, keeping everything else as it is defined on the template.
To do this, you can define partial formats. You define a partial format by adding one or more attributes to the name of the format. For example, if you name the format Header(background;font.color) only the font and the color of cell I10 will be applied, not the full format.
You can add as many properties as you want to be applied for the format separating them by semicolons (";"). Also, you can use negative properties by preceding them with a "-" sign. For example, the following definition: Header(All; -Border; -Font) will apply all attributes in cell I10 except the border and the font.

The list of properties you can write in a cell is the following:

Name	Description
All	Applies the all the formats in the cell. Defining a format named "header" is the same as defining one named "**header(All)**". You will normally use the All format when excluding formats. For example, "**header(All, -Border)**" will apply all the formats except the border. "**header(-Border)**" would not apply any format.
Border	Applies the four borders. "**header(Border)"** is the same as "**header(Border.Left;Border.Right;Border.Top;Border.Bottom)**"
Border.Left	Applies the left border.
Border.Right	Applies the Right border.
Border.Top	Applies the Top border.
Border.Bottom	Applies the Bottom border.
Border.Exterior	This is an special setting, used to apply the borders only on the outer bounds of the range. For example, if fmt is defined as "***fmt(border; border.exterior)***", and you write ***<#format range(a1:c5;fmt)>*** then the top row of the range (row 1) will be formatted with the top border format of the cell, the left column or the range (column a) will be formatted with the left border format of the cell, and so on. Inner cells in the a1:a5 range will not have any border applied. This tag alone has no effect; you always need to use it together with other border tag. For example "***fmt(border.exterior)***" will not do anything, you need to write "***fmt(border; border.exterior)***", "***fmt(border.top; border.bottom; border.exterior)***"or something similar. Other properties (like background) are not affected by border.exterior, they will still apply to the whole range. Note that this tag only applies to the <#format range> tag. It makes no sense in <#format cell>, and you cannot use it in <#format row/column> since format for

	columns and rows do not support this.
Font	Applies all the font properties. "**header(Font)**" is the same as "**header(Font.Family;Font.Size;Font.Color;Font.Style;Font.Underline)**"
Font.Family	Applies the font name.
Font.Size	Applies the font size.
Font.Color	Applies the font color.
Font.Style	Applies the font style (**bold** and *italics*) Underline in not included in Font.Style.
Font.Underline	Applies the underline.
NumericFormat	Applies the numeric format of the cell.
Background	Applies the background color of the cell. "**header(Background)**" is the same as "**header(Background.Pattern;Background.Color)**"
Background.Pattern	Applies the fill pattern for the cell.
Background.Color	Applies the fill color for the cell.
TextAlign	Applies the horizontal and vertical alignment in the cell. "**header(TextAlign)**" is the same as "**header(TextAlign.Horiz;TextAlign.Vert)**"
TextAlign.Horiz	Applies the horizontal alignment in the cell.
TextAlign.Vert	Applies the vertical alignment in the cell.
Locked	Applies the "Locked" attribute in the cell.
Hidden	Applies the "Hidden" attribute in the cell.
TextWrap	Applies the Word Wrap setting for the cell.
ShrinkToFit	Applies the Shrink to fit setting.
Rotation	Applies the Rotation.
TextIndent	Applies the indent.

Important Note: You need to specify what the format applies to in the format definition, but not when calling the format. You might define a format as header(border), but you need to call it with <#Format Cell(header)>, not <#Format Cell(header(border))> The second way will not work.

With partial formats you can apply the different formats as "layers" in your document, one independent from the other. You can apply as many <#format cell> and <#format range> tags as you want in a cell, as long as the applied formats are different all of them will be applied.

A last word of caution: As powerful as custom formats are, remember that they are only useful if you want to conditionally format cells. If the format is static, just format the cell as you want it. We have seen too many customers "over engineering" reports with lots of format cell tags, when the simplest solution would be to format the cells directly in the template with Excel and not use any tag.

You can see an example on how to use custom formats to have alternating rows in the "Mutiple Sheet Report" demo.

Grouping Tables
Sometimes you might have the data for a master detail report into one table, and you want to create two different tables based on the value of a key field.

You can use the "DISTINCT" filter on the config sheet together with the "RELATIONSHIP" tag to get this effect. Look at the "Master detail on one table" demo for more information on how this is done.

Note: Whenever possible, do not use this grouping as the normal way to get data. While it simplifies the data layer, it also fetches a lot of repeated information from the database.

For example: If you

```
select * from table1 join table2  on (table1.key = table2.key)
```

you will get results like this:

```
table1.value1  table1.value2  table2.value1  table2.value2
table1.value1  table1.value2  table2.value3  table2.value4
table1.value1  table1.value2  table2.value5  table2.value6
```

On this simple case, you fetched 12 values from the database. If you had made 2 different selects, you would have fetched only 8 values (table1.value1 and table1.value2 for the first table, and table2.valuen for the second). Depending on the amount of data, there might be a lot of repeated fields on the join.

Splitting Tables
The same way sometimes you might want to group tables, other times you might want to split them in groups of n records. For example, you might want to create a 5 column report, and you need to split the master dataset in groups of 5 records in order to fill the columns.

This is where the **Split(source, number of records)** tag can be useful. In short, you write this tag on the "Source Name" column in the config sheet, specifying the table to split on the **"source"** parameter, and how many records you want on each group on the **"number of records"** parameter. This will create a new table that you will name on the "Table Name" column of the config sheet. You can then use this new table as master on a master-detail relationship with the original table. The generated master has no columns, but you can use the pseudocolumns (#RowPos or #RowCount) just fine. Each record of the master is related to "number of records" records on the detail.

Note that the generated master table is a “pseudo table” n the sense that it has no columns or data, but it has (DetailRecords.Count – 1) / NumberOfRecords + 1 rows. Also the relationship between the master and the detail is not on real columns, since there are no columns on the master. This creates a limitation on how you can use those tables, and it is that the master should have the detail as a direct child. You cannot have other __ range__ between them, or FlexCel will complain.

Take a look at the Split Datasets demo for more information.

Retrieving TOP N Records from a table
You normally should filter the data when retrieving it from the database (for example with the SQL “Select top 10 * from customers”). But if this is not possible, you can use the Top(source, number of records) tag to filter this from the template. **Be careful with this tag.** If your table has 10,000 records and you only need 10, fetching them all from the db in order to use only 10 is not a smart idea. Take a look at the Fixed Forms With Datasets demo for more information.

Direct SQL on templates
Depending on your needs, you might want to write SQL commands directly on the templates and avoid having a Data module on code. This allows the users to modify the data they need by modifying the xls file, and without recompiling the executable.

But, before you continue reading, be aware that allowing your users to directly write the SQL commands can mean a big **security risk.**

For example, a user could use the connection you give him to execute the SQL: “drop table users” instead of a normal select. While FlexCel does a little validation on the SQL written by the user (for example, it cannot contain “;” or “--”, it has to start with “Select” etc) SQL is a very powerful language and there can always be a way to execute a command on the server. And, even if the user does not manage to execute a command, he might always do a “Select user, password from users” or similar command, and get dbadmin access to the database.

So, you must supply a readonly connection, and with rights limited to the tables you want the user to see.

The steps for allowing Direct SQL on the templates are:

1. Provide a connection to the report, by using FlexCelReport.AddConnection()
2. On the config sheet, “Source Name” column, add a string like “SQL(Select * from clients”). Give this table a name, and you can use it as any other table on the report.

Note that also for security reasons, you can't replace expressions inside the SQL string. For example, you can't write “SQL(select * from customers where cust_id = <#custId>)” This would open another security hole and allow for SQL injection attacks.

SQL Parameters

You need to specify database parameters to be able to actually pass information to the SQL.

.NET can use both positional (“?”) and named (“@name” or “:name”) parameters, and some data providers will accept one or the other. As we want to keep the template database independent (so you can replace the db backend without changing templates), **all parameters on the template are**

named, with a preceding "@" ("@name"). When using a db that needs positional parameters (like ADODB) or has another syntax for named parameters (like oracle that uses ":name") the SQL will be automatically converted by FlexCel into the needed one before sending it to the db.

FlexCel includes support for the most common cases, and when it does not know which db it is, it will "guess" that the db uses named "@" parameters.

When you use a db FlexCel does not recognize (for example Oracle), you need to let FlexCel know which kind of parameters your db needs. For this you use the **SqlParameterReplace** and **SqlParametersType** properties on FlexCelReport. (look at the reference help on those properties to see how to use them)

You can see a demo on how to do it on the "Direct SQL" example; more information is also available there.

User Tables defined on the Template

Sometimes Direct SQL is not an option, because you have your own data layer that you want to use, or because it is too big security risk to let your users run arbitrary SQL commands. But you would like the advantages of Direct SQL. That is, to specify the data directly on the template, so everything is self contained on the xls file and you can change your reports without recompiling the application.

You can use the **USER TABLE**(Params) tag to achieve this.

User table(Params) is a very simple tag, but it allows a lot of things. You write it on the Source Table column in the config sheet, and you can add an additional parameter on the Table Name column. You can leave the "Sort" and "Filter" columns empty or with values, their values do not matter.

For every "User Table(Params)" on the template, the event UserTable will be called on the report. Anything you write in "Params" will be passed as arguments to the event, without any further processing by FlexCel. Also, the value you write on the Table Name column will be passed to the event. You will normally want to use this second parameter to tell FlexCel how you want to name the table you are creating. Note that the name you write on the "Table name" column is not guaranteed to be created; this all depends on what you write on the event handler. Also, note that you must write something on the tablename column even if you will not use it on the event, or the row will be ignored.

So, how do you use it? Imagine you write a tag: User Table(CUSTOMER), and define the event UserTable so each time it gets the parameter "Customer" it loads the customer table from the database and adds it to the report. This way you are actually telling FlexCel which tables it needs to use on the template, allowing you to add new ones (from a list of available tables on the application) or remove existing tables without changing the code.

Other way to use this tag could be if you have your own API to access the database, with your defined commands, permissions and validations. You could pass the API command as parameter on the <#User Table> tag, and use this parameter inside the event to execute the API and add the generated table to the report. You would write something like <#User Table(get table customer on customerId < 100)> on the template, and on the UserTable event on the report, execute the parameter against your API, and add the resulting dataset.

Note: When using the "User table" tag to pass arbitrary API commands to the application, please remember to validate permissions on the UserTable event to see if the user is allowed or not to run the query. Forgetting to do so could generate a big security hole on your application, the same way as the SQL tag could.

For more information on this tag, see the User Tables demo.

Debugging Reports

Introduction

Whenever you hit enough complexity in your reports, you will get expressions that do not behave as you expect, and you will need tools to investigate what is really happening under the hood. This chapter speaks about those tools.

FlexCel Reports are declarative instead of imperative. This means you describe what you want using report tags, but you do not write code to tell the computer how to do it. In a way, FlexCel Reports are similar to SQL, and different from normal code.

```
XlsFile xls = new XlsFile();
xls.NewFile();
xls.SetCellValue(1,1,"Table");
if (Table["test"] == null) "error"; else
xls.SetCellValue(1,1,Table["field"]);
```

A "declarative" report would be a template with the text "Table" in cell A1 and the tag

```
<#if(Table.Test =;error;<#Table.field>)>
```

in cell A2.

Declarative reports are normally more "resistant" to bugs because they are simpler and so it is easier to see what is wrong. But on the other hand, they are also more difficult to debug when there is a bug, because there is no code you can step with a debugger.

In the XlsFile example above, you could set a breakpoint in the last line and evaluate the values of the variables before execution. In the FlexCel Report there is no way to set a breakpoint, since there is no code to execute. But you can still find out what is going on, and this is what we will explain here.

There are two main causes for errors in FlexCel reports; syntax errors and logical errors. They are covered below.

Dealing with Syntax Errors

Those are the easiest to deal with. The same way a "Code" report will not compile if there is a syntax error, a FlexCel report will raise an Exception when "compiling" the template if it finds anything it cannot understand. The error message will normally tell you what is happening and where, and there should be no big problem in fixing it.

If for example you write "<#tag" in a cell, you will get an error telling you that there is a closing tag ">" marker missing.

Now, in some situations it might be useful to see all syntax errors at once, and for this FlexCel offers a "ErrorsInResultFile" mode. In this mode, all errors related to tags will be written to the cell where the tag is, instead of raising exceptions, and the report will continue to generate. Errors will have a yellow background and red text.

	A
7	*Value*
8	ERROR: Report Variable "invallid" is not defined.

This mode does not cover all errors (for example a named range for a non existing table will still raise an error), but it covers the most common issues. The others still need to raise an exception, since if for example you have a range named "__table__" and no "Table" in the report, there is no place where FlexCel could write this error inside the xls file. The error must be in a cell for this mode to work.

There are two ways to enter ErrorsInResultFile mode. The first one is to change it in the code before running the report, by changing the ErrorsInResultFile property in FlexCelReport.

For example:

```
FlexCelReport fr = new FlexCelReport(true);
fr.ErrorsInResultFile = true;
fr.Run(...);
```

The second way is to write it directly in the template, inside the config sheet.

For example:

	M	N
8	Expressions	
9	Name	Expression
10	<#ErrorsInResultFile>	

The <#ErrorsInResultFile> tag can be anywhere in the Expressions column, and you do not need to write anything in the Expression definition.

This second way to enter ErrorsInResultFile mode is better when you are editing a template and want to do a debug without modifying the code, while the first way is better if you are automating testing and do not want to modify the templates.

Note: Remember that this mode is a "debugging" mode, and you should turn it off for production. You do not want to ship a file containing error messages in cells to your customers.

Dealing with Logical Errors

These kinds of errors are harder to deal with, are more subtle, and can pass without notice since they do not raise any error on FlexCel side.

For example, imagine that you have this expression:

```
<#if(<#tagval>=<#refval>;OK;Error)>
```

This is a valid tag, and the report will compile and run without issues. Let's imagine now that tagval is 1, and refval is “l” (lowercase L). So, when tagval is 1, the condition will evaluate to false, and you will get the “Error” label instead of “OK”.

With this font in particular this can be a hard to spot problem, because as we said before, you cannot really put a breakpoint in the expression and see what is inside tagval or refval.

Here is where the “**Debug**” mode can help.

As with the “**ErrorsInResultFile**” mode, there are two ways to activate debug mode. The first is by code, by setting the “**DebugExpressions**” property in the FlexCelReport to true. Again, setting it in code is useful when automating tests because you do not have to change the template.

And the second way is to write “<#**Debug**>” in the configuration sheet, the same way as in ErrorsInResultFile. This second way is preferred in most cases, f.i. if you do not want to modify the code, for example when testing a template.

The same remarks mentioned for the <#ErrorsInResultFile> apply to the <#Debug> tag.

In this mode, tags will not write their value to the cell, but they will rather write the whole chain of calculations made to arrive to the value.

For example, for our problematic expression:

```
<#if(<#tagval>=<#refval>;OK;Error)>
```

We will get this result in the cell:

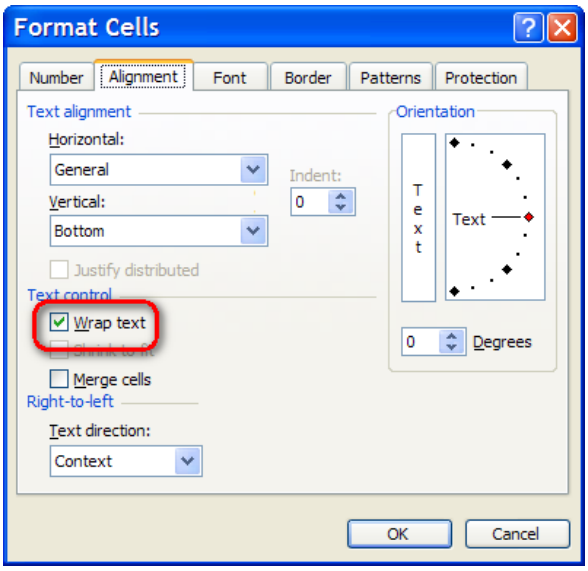
	B
9	**if(<#tagval>=<#refval>;OK;Error):** *Error* **refval:** *l* **tagval:** *1* **<#tagval>=<#refval>:** *False* **Constant:** *Error*

The format of the lines is as follows:

Each line has an expression in bold and a value in italics. Under that line and indented to the right we find the subexpressions used in the main expression and their values.

In this case, we can easily see that <#tagval> is “1”, <#refval> is “1”, and <#tagval> = <#refval> evaluates to false, so there is something wrong in the test. Having the full stack and all the intermediate values used to compute the main expression can be a valuable tool to find out what is going wrong, by allowing us to easily locate the exact point where the expression is not evaluating as it should.

Note: In order to correctly visualize the stack in the cell, you will have to set "Wrap Text" to true in the cell properties. If Wrap text is false, all lines will show in one line and it will be more difficult to read.

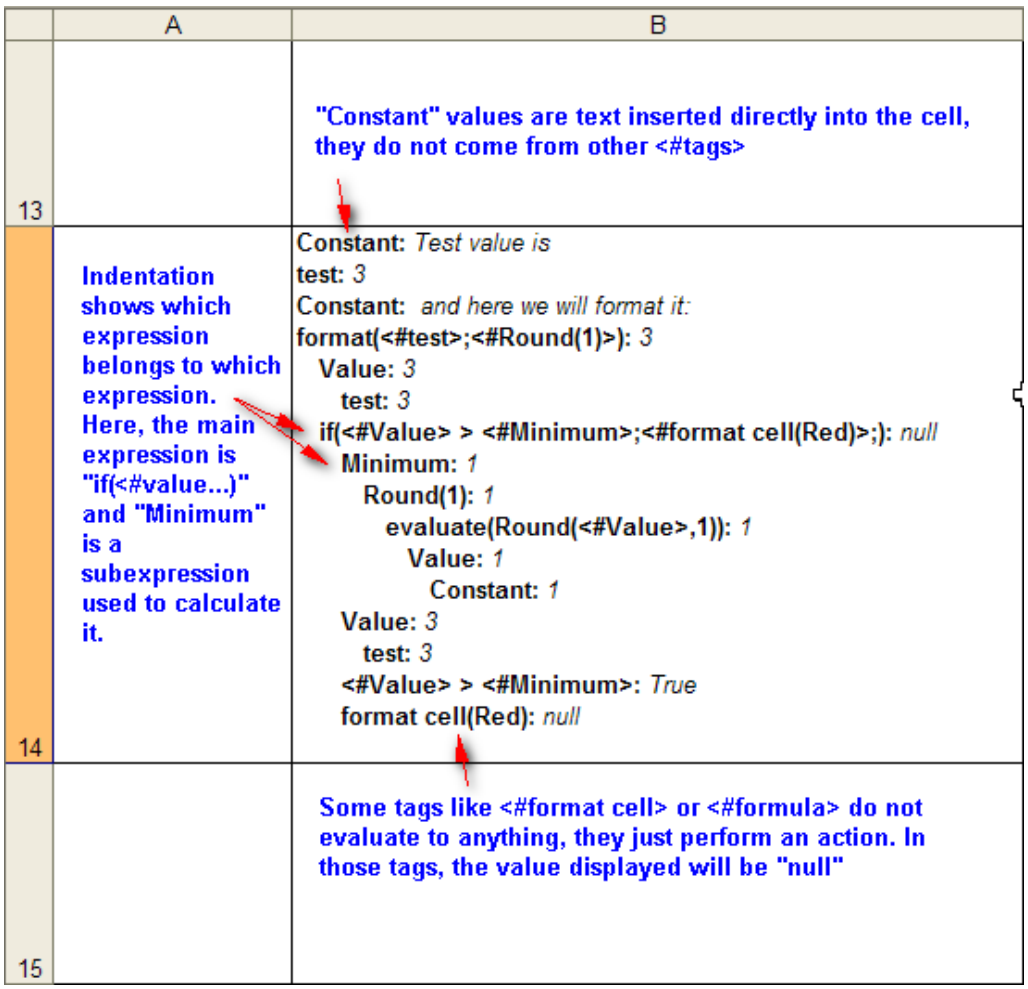


Understanding the stack can be a little difficult at first time, but once you get used to it, it can be a great help. To end up this chapter, we will present a more complex expression and its corresponding stack explained. You can see it yourself at the "Debugging Reports" demo.

The original expression is this:

	B
14	Test value is <#test> and here we will format it: <#format(<#test>;<#Round(1)>)>

And the result we get when running this template in "debug" mode is:



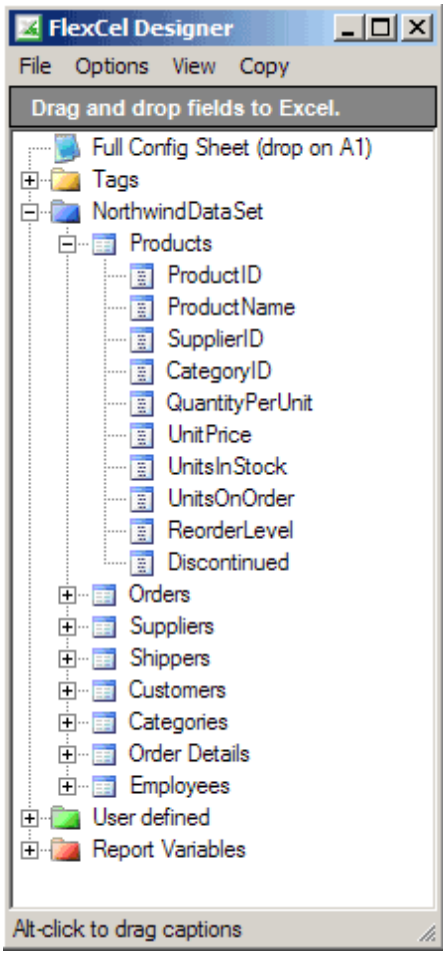
You can see we have 4 main expressions here (they show without indentation). The first one is the constant text "**Test value is**", the second is the tag "<**#test**>" that evaluates to 3, the third is other constant text "**and here we will format it**: " and the fourth is an expression "**Format(<#test>;<#round(1)>)**" that evaluates to 3. So the result in "normal" mode of this cell will be "Test value is 3 and here we will format it: 3". As there is a "format cell(red)" tag in the evaluation stack, the result will be formatted in red format.

All tags shown below the "format" line are subexpressions used to compute it.

Appendix: Using FlexCel designer

While you can create your reports only with Excel, writing all the tags can be a time consuming task. FlexCel designer is a little tool created to help on this.

When you launch FlexCel designer, a screen similar to the following will open:



If no file is loaded, the listbox will only contain “Full Config” and “Tags” entries.

Screen description

On the listbox you can select a field, a tag, a report variable or a full configuration sheet, and drag and drop it to Excel or Open Office.

If you press “Alt” before beginning to drag a field, the field and the field name will be pasted on Excel.

Menu description

The following options are present on the menu:

File - Open:
Will open a new report. It will read the report variables and dataset description file from the configuration sheet on the report, and display them on the listbox.

The last file open will be saved when you exit FlexCel designer, and will be restored next time you open it.

File - Exit:
Will close the application.

Options - Use Column Captions:
When checked, the column captions instead of the column names will be dropped on the report. When using this option, make sure the corresponding FlexCelReport has UseColumCaptions property = true.

View - Always on top:
Will maintain the window on top of the others, so it is easier to drag to Excel.

View - Opacity:
Sets the transparency of the form. Works only on Windows 2000 or later.

Copy:
Will copy the selected item to the clipboard. This can be useful when editing composed statements (for example <#d1> <#d2>) because you can't drag and drop to the middle of a cell. But you can copy/paste <#d2> anyway.

Data configuration

FlexCel designer loads its data from the configuration sheet of an xls file. If the report does not have a configuration sheet, you can't open it with FlexCel designer.

On the sheet, it will read:
• The xsd dataset definition file, and add the dataset to the listbox. The xsd file is generated by Visual studio when you edit a dataset schema. You should copy this file to the same folder the template is in.
• All the user defined tables, and add them to the user defined folder.
• All the user defined variables, and add them to the report variables folder.