

Having it your own way with ODS

Frank Poppe, PW Consulting, the Netherlands

INTRODUCTION

The Output Delivery System (ODS) of SAS® makes it possible to influence the output of an application in a great many ways:

- the format in which it is delivered (HTML, PDF, etc.)
- the organisation of the output
- what exactly goes where
- how it looks like in general (the style)
- change several aspects of the above for particular items depending on the data.

This is both true for the procedures developed for customised output (REPORT and TABULATE in particular) and for the other output producing procedures (often used procedures like MEANS, FREQ, UNIVARIATE but also all the others).

However, it is not always clear where to look when one has to change the output in some way. In this paper I first want to treat the different elements of ODS, and the way they together determine the final output. I then will describe how one can change and enhance the behaviour of these elements.

I then want to go on to describe how one can link different parts of the output together. This will focus on HTML output.

The actual presentation will contain live demos as much as possible. In this summary on paper you'll have to with some screenshots. Some of the code I use however can be downloaded to serve as a starting point for your own endeavours in the world of ODS.

THE ODS ELEMENTS

What happens behind the screens when a SAS procedure or data step produces output? Different parts of SAS, and in particular ODS, come into action, acting together or one after the other, to produce HTML, RTF or LaTeX, or whatever. This can be written to a file, or directly to a webserver, which will pass it on to someone's browser, or as an e-mail application to one or many addressees, to mention a few possibilities.

What I will describe here are the elements in the way I as a SAS programmer see them. What really exactly happens may be slightly differently, but I think this is practically speaking not very far of. I've based this description on the 'official' documentation (as laid down in the OnLineDocs), on the less official information that can be found in the *Base Community* pages on the SAS website, and on e-mail correspondence with some of the SAS developers that maintain those pages.

And of course on lots of experiments, trying to discover why things didn't work out the way I initially expected...

The output destination

The output destination is definitely the best known part of ODS. Destinations like HTML, RTF and Printer were with us from the beginning of ODS. Since then the ODS team generalised the concept into the Markup Language. Through that route a lot of other destinations were introduced, some for new 'markup languages', some as variants for existing ones (like compact HTML, e.g.).

A lesser known, and certainly lesser used, fact is that you can also use this concept to define new destinations, through the DEFINE TAGSET statement of PROC TEMPLATE. I'll show an example of that later in this paper.

The output stream

As soon as you use an ODS statement for a destination you create what I call an *output stream*. Most destinations take a lot of other options, but you will always have to define the physical target.

The physical target will often be a file, or a *fileref* pointing to a file. But that *fileref* can point to anything SAS can write too. The automatic *fileref* _WEBOUT e.g., will direct the output to a web browser (if you have your webserver configured). But you can also have a *fileref* pointing to an e-mail application that will send your output immediately to your clients as soon as ODS CLOSE is executed. Or you can have a *fileref* maintaining a DDE link to another application, which will write your ODS output directly into that other application (providing the application understands the 'language' of the destination).

The output object

When you have your *output stream(s)* set up, by pointing a destination to a *fileref*, what happens when the procedures executes?

Before ODS entered the arena, all SAS procedures generated the output in their own way, as ASCII text into the listing, or as one or more SAS data sets. Now all SAS procedures offer their data to ODS. (Also the data step can write to ODS, but I'll not go into that here. There isn't much fun there at the moment, I think. But that will change in SAS 9.1)

The data is offered to ODS as raw data, but the procedure also has to tell how to turn that data into a table. Specifying a table definition does that.

The table definition

In general procedures take the table definition from the set available in the ODS template stores. There are only some procedures that generate their own table definitions. Within the ODS community they are sometimes called *the gnarly four*: PRINT, REPORT, TABULATE and FREQ. These procedures do not output standard tables; on the contrary, it is the very purpose of the syntax of these procedures to define the layout of the tables.

There are lots of table definitions inside ODS. When people look into the templates supplied by SAS (by right clicking on the Results in the Results Window and choosing Templates) they usually go directly to the *Styles* folder. It can be worthwhile however to have a look at all the other folders as well. If you (or your client) e.g. would like to change the header "Mean" in table of simple statistics from PROC CORR into "Average", you only would have to change that header in the Base.Corr.UniStat table template. A step further would be to introduce symbols like σ , μ and Σ for Std Dev, Sum and Mean. That can be done too, but is a bit more difficult: often the code to get that symbol depends on the destination you are working with. On our web site you can find a macro which handles that (<http://tips.pwcons.com/modules.php?name=News&file=article&sid=88>). Sometimes you have to take special care to get special symbols like $\&$, $<$ and $>$ through to the final target without changes. Another obvious thing to do would be to change the default format for some columns. Changing table definitions can be done with the DEFINE TABLE statement (and some of its assistant statements) of PROC TEMPLATE.

The style

With the *table definition* ODS knows where to put the raw data from a procedure in a table. How that table and its cells will look is determined by the ODS style that is chosen. This is one of the options when you open an output stream for a particular destination. If you don't choose, SAS will choose a default style.

I have mentioned a few DEFINE statements of PROC TEMPLATE already, but the DEFINE STYLE statement of that procedure is probably the one people are most accustomed to. Nevertheless, defining a new style or changing some aspects of an existing style can offer surprises. I'll treat some of those in an example later on.

Bringing it together

I have tried to make an overall view of the elements I have described and of their relations. I am not sure it adds something in itself, but maybe with the above explanation it gives some more insight. You can find in the appendix.

CHANGING STYLE

The most popular use of ODS is for changing the style of the output. I look at it here from viewpoint of having to define a more or less complete corporate house style, in contrast to changing just one column of one table from a particular procedure.

Using existing style sheets?

The first thing to think about when you want to apply a particular style for your SAS output is whether you really want to use SAS to implement the style. If you are generating a lot of HTML for a company that has already implemented its style in a style sheet (a CSS file), it might be easier to use just that. Tell ODS to use an existing style sheet, in the following way. Instead of

```
ODS HTML file = 'C:\path\r.html' stylesheet = 'C:\path\r.css' ;
```

specify

```
ODS HTML file = 'C:\path\r.html' stylesheet = ( url = 'X:\alt\comp.css' ) ;
```

This way you do not specify a target file for the style sheet, the link to an existing style sheet that will be written into the HTML file.

But often it isn't as easy as that. ODS uses HTML *classes* to select which particular style to use for a certain element (a cell header, a data cell, a title, etc.). Maybe your company style sheet uses classes as well, but they

will have different names. So you then will have to describe the link between the two, which mean you are still defining an ODS style, but in terms of HTML classes instead of in terms of attributes like fonts and colours.

The situation changes when you have to provide output for different destination, e.g. not only HTML but also RTF to include in a Word document, or PDF for distribution. Even if there does exist an implemented style for one or more of those destinations, it might be easier to redefine them in a single style within ODS. Any changes and enhancements then can be made in a single place, and will find their way to all the output destinations. Also ones you don't use right now (or might not yet exist in SAS), but might be requested later.

To be honest, it probably will not be one single place. The style for output on screen will always differ from a style for output on paper (at least, I think it should). But they will have a lot in common, and you can have them share those common parts.

That you *can* share common parts is because the styles know about inheritance. In fact, the default styles rely very much on inheritance. Convenient as this often is, it can result in unexpected results. The following paragraph goes into that.

Inheritance

Let us look at a small part of the definition for the default style.

```
style HeadersAndFooters from Cell /
  font = fonts('HeadingFont')
  foreground = colors('headerfg')
  background = colors('headerbg');
style Cell from Container ;
style Container /
  font = Fonts('DocFont')
  foreground = colors('docfg')
  background = colors('docbg');
```

The syntax `style <new-style-element> from <other-style-element>` is very common. In this way style elements *inherit* all attributes from other elements. *HeadersAndFooters* inherits from *Cell*, *Cell* inherits from *Container*, which is the starting point for everything.

Another aspect to note is that the attributes shown here are not defined directly, but *by reference*. The references '*colors*' and '*fonts*' can be seen as arrays, and they too are defined in the style definition. There the elements '*docfg*' and '*DocFont*', etc., are given real values like `Black` or `Arial`.

So in the default style both *Data* and *Header* inherit (indirectly) from *Container*, but for *Header* three attributes are changed, and for *Data* two of them.

There are about 100 style elements to define. ODS will call upon those elements to specify the look and behaviour of the different elements in the output, so you have to make sure they are there. You must think here of elements like *table*, *data*, *dataFixed*, *header*, *headerStrong*, *systemTitle*, *byLine*, etc., etc.

The inheritance feature, and the hierarchy it produces, makes it easier to maintain and change styles. By changing attributes in an appropriate place in the hierarchy, one can force that change to trickle down to all elements inheriting from the changed element. Therefore the *default* style is the preferred one as a starting point. However, this is not without practicable problems. Some arise from the fact that the default style has very long chains of inheritance, which not always are intuitive.

It also should be kept in mind that all HTML output in the end are *cells* that are put within *tables* (even the titles we have seen already are one-cell tables). Tables and their cells have their own attributes, and browsers have their own ways in combining those attributes.

In the following paragraphs I will try to show some of these subtleties, and ways to get around it.

The first try

Let us say I want the overall background of the output changed. This now is a light gray, and I want it changed to a picture (this may not be the best result for easy reading, but makes it easier to show the differences in black and white output).

We cannot change the styles in SASHELP, so we define a new style (let's call it *NEW*), and define the existing style as its *parent*. That means that all elements not explicitly defined in our new style will be taken from the *parent* style.

The general appearance is done by the *Body* element, so we set the attribute for that object. The job and a screen dump of the result are shown.

```
PROC TEMPLATE ;

  DEFINE STYLE styles.new;
    PARENT = styles.default ;
    STYLE body /
      backgroundimage = "logo.gif"
    ;
  END ;
```

This is what we had specified, but is it what we wanted?

It would be nicer if the tables wouldn't use their own backgrounds, but also would show our graphic. Note that this illustrates too that the title is also presented inside a one-cell table.

The hierarchy shows that all elements inherit from the abstract element Container, in the 'root' of the hierarchy. What if we redefine that element, so that this trickles down through the hierarchy? It might be a solution to put the backgroundimage definition there.

Obs	Name	Sex	Age	Height	Weight
1	Alfred	M	14	69.0	112.5
2	Alice	F	13	56.5	84.0

One might think that the following style statement would do that trick.

```
STYLE container /
  backgroundimage = 'Logo.jpg'
;
```

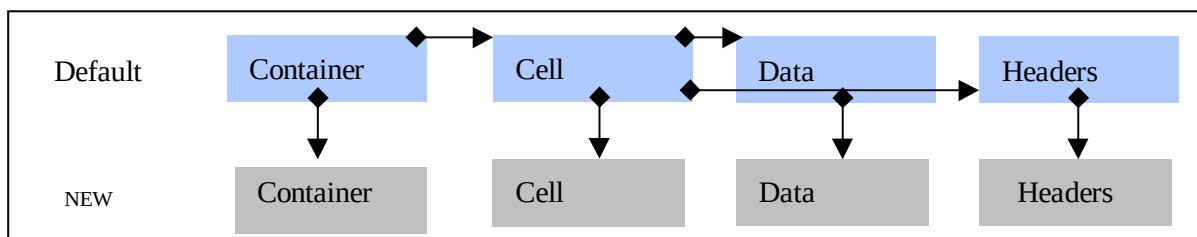
To explain why this will not work we must take a closer look at the inheritance of style attributes when modifying an existing style.

Hierarchy and inheritance

How does a new style take its values from the old one?

We have seen already that we are not actually changing the original style, but that we merely designated it as the 'parent'.

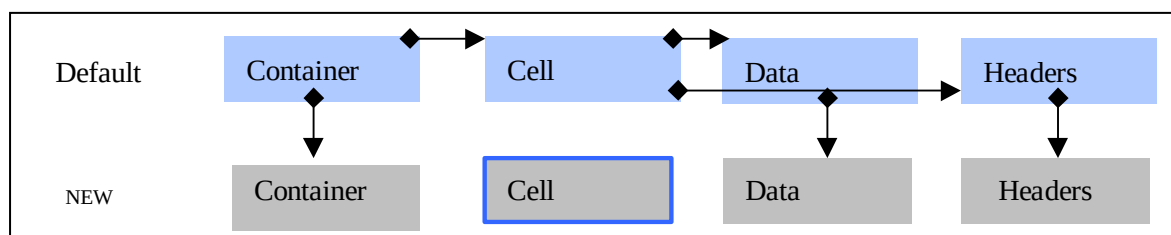
In the following simple scheme we look only at four elements: a *Container* defined at the 'root', a *Cell* inheriting from it, and *Data* and *Headers* inheriting from *Cell*. If we start defining a new style *SEUGI* a kind of virtual copy is created, were all elements are present too.



We now redefine an element in *NEW*, let's take *Cell*.

```
STYLE Cell / someattrib = newvalue ;
```

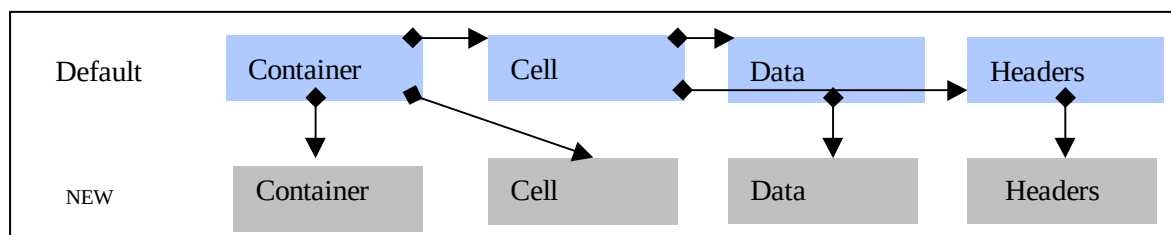
The link of that cell to the parent is broken, and a fresh copy is inserted instead. In the following picture we have shown that by giving it a border.



If you now follow the arrows, you will note that the *Data* and *Headers* elements still inherit from the *original Cell*, through the original *Data* and *Header* elements! Note also that the new element is not connected by any arrow to an old element, indicating that the new *Cell* element does not inherit any longer from *Container*. So any attributes that were defined in *Container* and not explicitly set in the new *Cell* are now undefined (or resort to system defaults).

You could add the 'From...' option to the definition to indicate that you want to use the old values from 'Cell' or from 'Container'. In the following the new 'Cell' inherits from the old 'Container'.

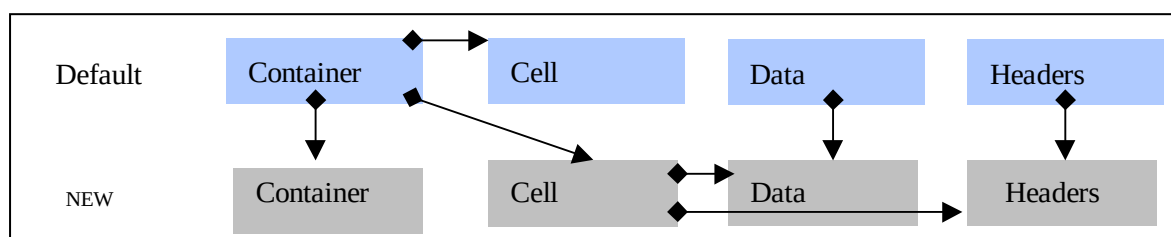
```
STYLE Cell FROM Container /
  some-attr = new-value
;
```



But what if you want the new 'data' and 'headers' elements to inherit from your new 'cell' element? For that you must not use the STYLE statement, but the REPLACE statement:

```
REPLACE Cell FROM Container
  "New definition" /
  /* my own specs for Cell */
;
```

Then the 'parentage' of the new 'cell' element takes over from the old 'cell' element:

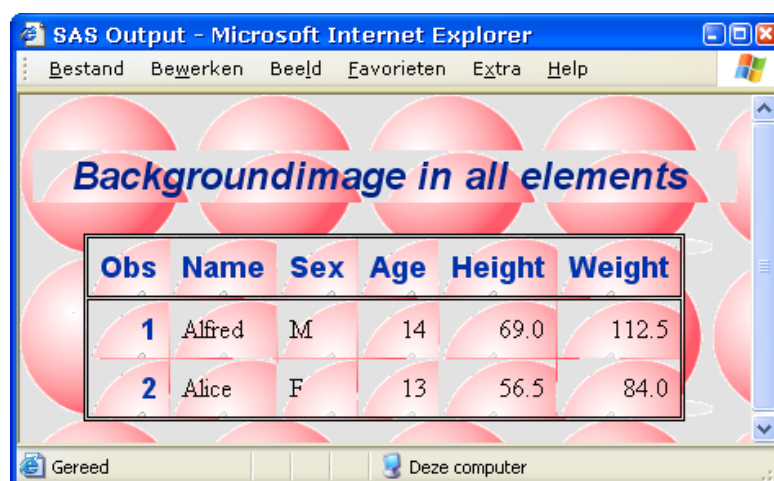


The parent will always be used whenever some inheritance takes place, *unless* you have used the REPLACE statement. So even when you define a completely new element, unknown in the parent style, and you want another element to inherit from that element, you *have* to use REPLACE (the manual incorrectly states that replace only should be used when replacing an existing element!).

So this now is the template definition, with the result next to it (if MS Word will not do one of its nasty tricks on the final printout...).

```
PROC TEMPLATE ;
  DEFINE STYLE styles.seugi ;
    PARENT = styles.default ;
    REPLACE container /
      backgroundimage = "Logo.gif"
    ;
  END ;
RUN ;
```

At first glance this looks nice enough, but if you look closer you will see that the background is repeated for each element: inside each table, and inside each element of that table the image is started again.



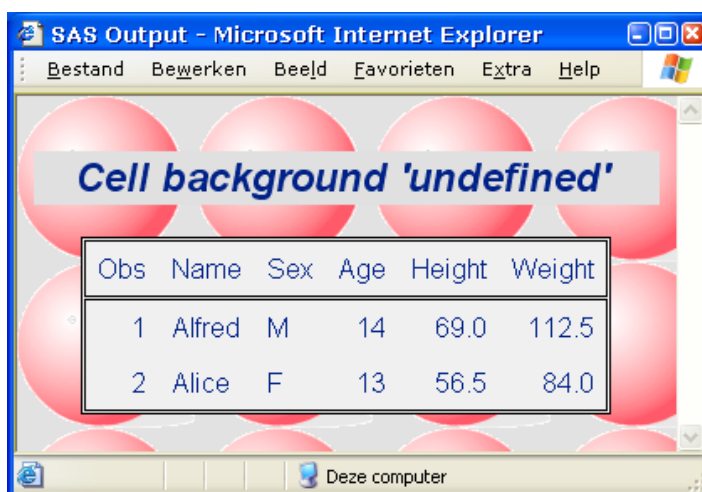
This is not what we wanted. So we have bind the background to the body, as we originally intended to do, and we want the cells to have 'no color'. For this ODS has the `_UNDEF_` value. There are three style-elements present as cells in this example, and we change those elements were they inherit there background color from.

```
PROC TEMPLATE ;
  DEFINE STYLE styles.seugi ;
    PARENT = styles.default ;
    REPLACE Body From Container /
      backgroundimage = "logo.gif"
    ;
    REPLACE Data FROM Cell /
      background = _UNDEF_
    ;
    REPLACE Header FROM HeadersAndFooters /
      background = _UNDEF_
    ;
    REPLACE SystemTitle FROM TitlesAndFooters /
      background = _UNDEF_
    ;
  END ;
RUN;
```

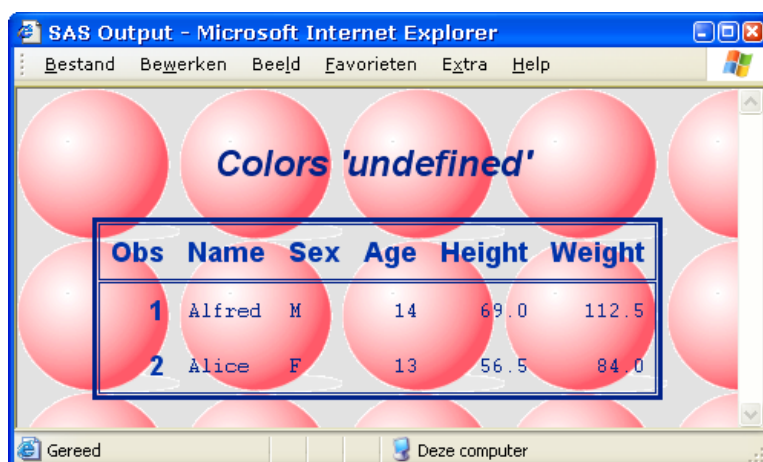
The result is unexpected (it was for me). A look at the HTML code revealed what is happening here: another type of inheritance comes into play. The cells do not have a background colour specified, but the table still has. And the browser let's the cell now 'inherit' from the table. So we also have to remove the colour from the table.

But if want to use this style for lot of jobs, there will be a lot places were the background colour has to be 'un-defined' (much more than the three cell-elements we have redefined already, plus the corresponding tables).

However, there is another route. We have already seem that proc template can handle 'reference lists'. A list of values (used in different elements through-out the style definitions) can be named and assigned values in the beginning of the definition, and used later. The original definition for the default style uses that concept to define a colour list. We can copy that definition, change it a little, and have the references in the original definition resolve to our new list.



```
PROC TEMPLATE ;
  DEFINE STYLE styles.seugi ;
    PARENT = styles.default ;
    REPLACE Body From Container /
      backgroundimage = "logo.gif"
    ;
    REPLACE color_list /
      'fgB2' = cx0066AA
      'fgB1' = cx004488
      'fgA4' = cxAAFFAA
      'bgA4' = cx880000
      'bgA3' = _UNDEF_ /* was cxD3D3D3 */
      'fgA2' = cx0033AA
      'bgA2' = _UNDEF_ /*was cxB0B0B0 */
      'fgA1' = cx000000
      'bgA1' = _UNDEF_ /* was cxF0F0F0 */
      'fgA' = cx002288
      'bgA' = _UNDEF_ /* was cxE0E0E0 */
    ;
    REPLACE Data From Container /
      font = fonts ( "FixedFont" )
```



```

        foreground = color_list ( "fgA" )
    ;
    REPLACE Table FROM Output /
        bordercolor = color_list ( "fgA" )
        borderwidth = 3
    ;
END ;
RUN ;

```

Additionally some final enhancements have been made. The *Data* cells have been given the fixed font from the *fonts* reference list, and the table has been given a stronger border.

ADDING LINKS

Reports often have different ‘levels’. First a general overview is given, and subsequently more and more detail on various aspects are given. This can mean a drill down into an organisation with departments, but the same is true for a report on a statistical analysis. Here the first level shows the overall statistics, and next the different χ^2 - and *p*-values for the variables are given. Or you just want to organise your information in different sections, and want to provide links from one section to the other. ODS makes it simple to set this up in a general way.

The following piece of code shows how you can organise your report into different sections and add drill down possibilities for further detail without even pre-processing the data.

```

ODS HTML
  &PathOption
  File = 'F.html'
  Style = sasDocPrinter
;
TITLE 'Females' ;

FOOTNOTE '<A HREF="M.html">to Males</A>' ;

```

I am again going to use the SASHELP.CLASS data set. The previous code initialises a HTML file titled ‘Females’, this time with style *SASdocPrinter*, just to show something else. A *footnote* is added referring to the companion HTML file for the males. The next code runs PROC REPORT selecting only the females.

```

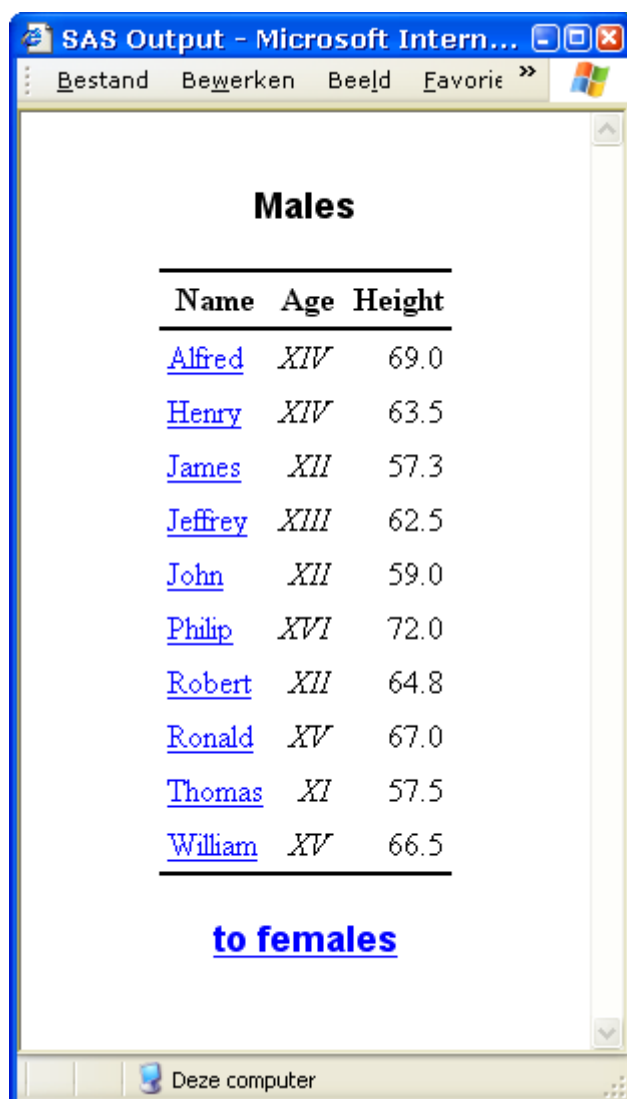
PROC REPORT NoWindows Data = sashelp.class (
  where = ( sex = 'F' ) ) ;
  COLUMNS Name Age Height ;

  DEFINE Age / format = roman. style(column) =
  DataEmphasis ;
  DEFINE Height / format = 6.1 ;
  COMPUTE Name ;
    Call Define (
      COL ,
      'URL' ,
      compress ( './img/ ' || name || '.jpg' )
    )
  ;
EndComp ;
Run ;

```

The DEFINE statements are fairly common, with perhaps as exception the format where I selected the ‘roman’ style for numerals.

The only ‘advanced’ work is in the COMPUTE block. Here the CALL DEFINE statement is used. The parameters of this statement are the following:



Males		
Name	Age	Height
Alfred	XIV	69.0
Henry	XIV	63.5
James	XII	57.3
Jeffrey	XIII	62.5
John	XII	59.0
Philip	XVI	72.0
Robert	XII	64.8
Ronald	XV	67.0
Thomas	XI	57.5
William	XV	66.5

[to females](#)

- COL, indicating that the statement acts on the current column (being NAME). For other purposes one could select e.g.. ROW.
- 'URL', indicating that the attribute to affect is the URL. Other attributes that can be changed in this way are (a.o.) STYLE and FORMAT.
- The new value for the attribute, in this case the URL, in this case constructed from the value of NAME. So 'Alfred' will point to the file ./img/Alfred.jpg (relative to the location of the HTML file).

This is enough for ODS to generate links in the HTML it produces.

The result shown on the right.

Equivalent code produced the page for the males, with a link to the females. With a little effort this can be generalised, without having to specify the different categories (*male* and *female*) explicitly. Usually some macroisation of the code is necessary.

LINKS AND GRAPHICS

Also graphics can have links. For that an HTML page with an image (GIF, JPEG or PNG) has to contain an *image map* linked to the image, describing the *hot spots* inside the area the image fills. All the SAS/Graph procedures can generate that map, and when you have an ODS destination open that code is written into the file automatically.

The way to instruct a SAS/Graph procedure to do that, is by specifying a variable that contains the URL to link to from that hotspot. For a choropleth map the statement would be something like

```
CHORO var / html = yourHTMLvar /* other options... */ ;
```

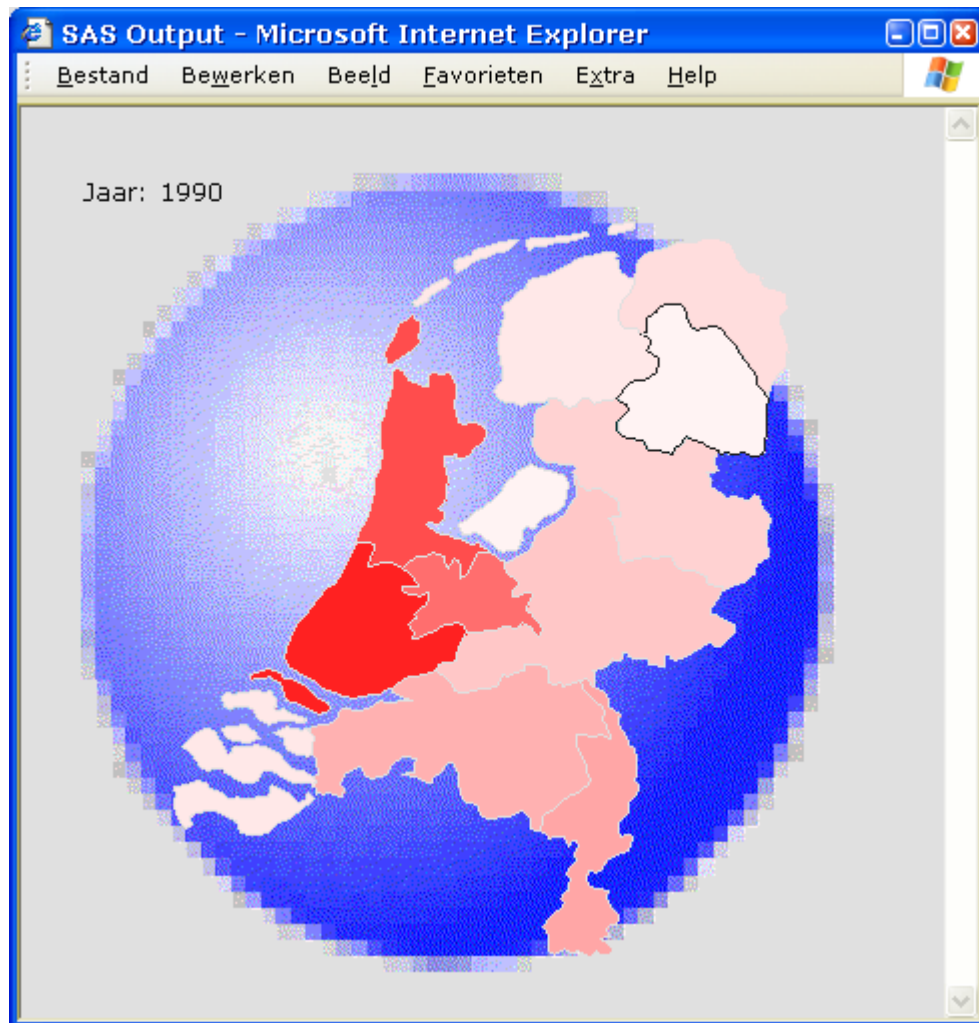
where *yourHTMLvar* contains values like "href=www.xyz.com/detail-123.html".

It is even possible to define hotspots for an animated GIF, although you have to trick SAS there. Of course this makes only sense if the boundaries of the hot spots do not change for the different pictures in the animation. The easiest way to produced an animated GIF's is to use the GIFANIM device driver, and using the chosen graphic procedure with a BY statement. Then append a '3b'x to the output (the hexadecimal value indicating the end of the animation sequence). But you do not want the *image map* written into the HTML file for each iteration of the BY value. To prevent this make first make a normal GIF from the data (e.g. for one of the values of the BY variable).

Then *close the HTML destination!* The HTML now contain a reference to the GIF file together with the image-map. Now you replace the GIF file by the desired animated one, by using only the LISTING destination. As I can not show an animated GIF in a word document, I can show here only a static example. What you see are the twelve provinces of the Netherlands, the colour codes indicating the population level. The screenshot gives the values for the year 1990, the animation shows other years as well.

I probably have overdone the use of appearance options, by adding also a background (goptions iback = ...).

Each province is clickable. The thin black line in the north-west indicates I clicked the province of Drenthe. The URL's I added here point to the web sites of the provinces.



CREATING A NEW DESTINATION

All the examples given above write to the destinations delivered with the SAS system. But since the introduction of the tagsets you are no longer confined to that. You can create you own destinations. Think of the creation of a file by a sequence of events, with many events having a separate beginning and ending:

- start new document
 - start table
 - start row
 - start cel
 - write value
 - end cel
 - start cel
 - write value
 - end cel
 - end row
 - end table
- end new document

The tagset syntax enables you to define exactly what to write for each event, where necessary for start and end of the event differently. For the HTML destination the 'start cell' event writes <TD>, and the 'end cell' event writes

</TD>. To be true this is an enormous simplification because in reality a lot more is written to take care of the attributes like font and colour.

When I happened to learn about these new possibilities I was working on a SAS application that should be able to write its output toward Excel. Nothing much was specified in what form or way, but it was clear that the less conversion and action from the users would be needed, the better the application would be received. I knew it was possible to write directly to Excel from a SAS application using OLE or DDE.

These two concepts, writing a new ODS destination and writing into Excel more or less found their way to each other in my head.

Writing through DDE is accomplished by defining the right *fileref*. Once that is in place you can send Excel macro commands from SAS to the worksheet.

In the end an ODS destination writes to a *fileref*. So would it be possible to make that a DDE fileref, and to define a tagset containing the right Excel macro commands? That would mean that any SAS application could direct any output from a SAS procedure to an Excel worksheet. It would even take into account style (fonts and colours) as far as I would be able to translate them into the right Excel macro syntax.

My contacts at SAS gave no answer. The DDE experts didn't know about ODS or tagsets or the way they write out their data. The ODS experts didn't know about DDE. So I just had to try, and by that time I was curious enough to do that.

And it worked (otherwise this chapter wouldn't be in the paper, of course...). Not without some subtle problems, but with the help of the above-mentioned experts I got it working.

I give here only some small parts of my code, just to give an impression. All of it can be downloaded from the SAS web site (http://support.sas.com/rnd/base/topics/odstagsets/#user_poppe).

```
define tagset Tagsets.Excel ;

  define event table ;
    start:
      put '[select("R[2]C1")]' NL ;
    finish:
      put '[select("RC2:RC")]' NL ;
      put '[border( , , , , 2 )]' NL ;
  end ;
  define event row ;
    start:
      put '[select("R[1]C1")]' NL ;
      put '[Border( , , 2 )]' NL ;
    finish:
      put '[Border( , , 2 )]' NL ;
  end;
```

This just gives an impression how you define a tagset.

The following piece of code shows how to use it.

```
FILENAME excelsys dde 'excel|system' ;

ODS tagsets.excel body = excelsys rs=none ;

TITLE 'ODS writes into your Excel sheet' ;
PROC UNIVARIATE data = sashelp.class ;
RUN ;

ODS _ALL_ Close ;
```

Excel should already be running when you define the *fileref*.

The solution works very well, but is not ideal. One problem is that you need to tell Excel at the beginning exactly what colours you are going to use, and you are limited to 63 different colours. So you have to know which colours the selected style uses, and there shouldn't be fancy code in the procedures that can select all kinds of colours (e.g. depending on data values). I have added some macros that make it easy to define a style in such a way that the colours will be known to Excel.

Fonts are a problem too. All the ODS styles in SAS define fonts using lists, first giving an exact font name, and then more generic alternatives, much like in (well behaved) HTML. This is a typical font definition in a SAS ODS

style: "Arial, Helvetica, Helv". Excel has serious trouble with these lists. The style has to give just a single font name.

CONCLUSION

In the previous paragraphs I have tried to give some examples of the various ways in which you can change and enhance the layout, the style and the format of the results of the Output Delivery System of SAS. What I have tried to show also is that often there are different routes you can take, and that is depends on your particular situation which is the most promising.

What I have shown here is based on SAS version 8.2. There is a lot more in v 8.2 that I have not shown here, not only because of lack of room, but also because I haven't explored them well enough. Now SAS 9 is arriving, with yet more possibilities. SAS is still developing, and some approaches are changing from version to version. That is partly because some of the new possibilities are not always as easy to use as the developers thought, because there are sometimes requirements they didn't think of. Therefore I want to encourage everybody to provide feedback: it has been my experience that the developers are willing to listen – although it is also clear that not each individual request will be honoured in the next production release of SAS.

ACKNOWLEDGEMENTS

Thanks to various developers of the ODS team in Cary and their colleagues in other departments for answering my different questions, and for their openness in discussing the 'technical problems' I sometimes encountered.

SAS and all other SAS Institute Inc. product names are registered trademarks or trademarks. Excel is a registered trademark of Microsoft Inc.

CONTACT INFORMATION

If you have any comments or question please feel free to contact me.

Frank Poppe
PW Consulting,
the Netherlands

Mobile phone: +31 6 21218860
E-mail: Frank.Poppe@Pwcons.com

APPENDIX: THE ELEMENTS OF ODS TOGETHER

