

Paper 225-2007

## ODS Markup, Tagsets, and Styles! Taming ODS Styles and Tagsets

Eric Gebhart, SAS Institute Inc., Cary, NC

### ABSTRACT

Find out how ODS styles can be used in new ways when they are combined with tagsets. Watch as a simple example grows into new ideas on how to use styles and tagsets. See how easy it is to create a new destination and style that do just what you need. Explore how these concepts have been applied to existing ODS Markup destinations, such as ExcelXP and HTML. Learn a few simple tools and tricks that will make styles easier to work with for any ODS destination.

Understanding just a little about tagsets and styles can enable the creation of those special reports you've always wanted. This paper reveals how styles and tagsets work, and how to apply that knowledge to various ODS destinations, or to create new and versatile ODS destinations of your own design.

### INTRODUCTION

The Output Delivery System (ODS) creates nice-looking output, but knowing how to use ODS styles and tagsets can create great-looking output. However, using style templates and tagsets can be intimidating and hard to understand. This paper begins with some simple examples that not only show how styles and tagsets work, but show how simple it is to make them work together to create great-looking reports.

### A COLLECTION OF STYLE ELEMENTS

The first thing you need to understand is what a style is. A *style* is a collection of elements (also called classes); each element has a set of attributes. These attributes define things like color, font, width, height, weight, and style. The elements that are defined in the style are used by ODS to affect the look of the various output elements in the report.

There are elements for title, footnote, BY line, table, data, header cells, and many other parts of the reports that are created by ODS. These elements can be seen best by using ODS to create output that you can look at. To begin, use a few special destinations that are made possible by tagsets.

Tagsets are another type of template that enable you to create your own ODS Markup destinations. There are a few destinations that are most helpful when it comes to understanding ODS and styles. For example, there are some tagset destinations that use HTML as their output type: Style\_Popup, Style\_Display, and namedHTML (this tagset destination is not discussed in this paper). The first tagset is called Style\_Popup. The Style\_Popup destination creates HTML that looks very much like the HTML that is created by the usual HTML destination, as seen in Figure 1. The difference is a special feature in the Style\_Popup output. Unfortunately, this special feature only works with Microsoft Internet Explorer. When you position your mouse pointer over an element on the page, the element turns a salmon color; and, if you pause long enough, a tooltip appears that shows the style name for the element.

Obs	Name	Sex	Age	Height	Weight
1	Alfred	M	14	69.0	112.5
2	Alice	F	13	56.5	84.0
3	Barbara	F	13	65.3	98.0
4	Carol	F	14	62.8	102.5
5	Henry	M	14	63.5	102.5
6	James	M	12	57.3	83.0

Figure 1: HTML Output from Style\_Popup Destination

If you click on that element, a pop-up window appears, showing the PROC TEMPLATE style definition for that element, as seen in Figure 2.

This style definition has no inheritance, so what you see is what you actually have, regardless of how the style template is written. Inheritance is one of the things that makes styles difficult to understand and manipulate. It is hard to see all of the things a style has defined because it might be inheriting a lot from its parent styles. This pop-up window is particularly handy because it shows everything a style has defined—it does not matter where it is inherited from.

One problem with the Style\_Popup destination is that it shows you only what you've done. So, if your job doesn't have BY processing, then there won't be any BY lines in your output, and you won't be able to see the style that is defined for BY lines.

There are many other elements that might not show up in your output based on the job that you run. To address that problem, there is another tagset, called Style\_Display, that creates sample output for all of the commonly defined elements. This tagset inherits its behavior from the Style\_Popup tagset, so it works the same way; but, at the end of your report, there will be a few pages of example output.


exampleA.sas shows how to list all of the tagsets, and how to use the Style\_Display destination.

#### exampleA.sas:

```
proc template;
  list tagsets;
run;

ods tagsets.style_display file="exampleA.html";
proc print data=sashelp.class;
run;
ods tagsets.style_display close;
```

The output that is generated by PROC PRINT is displayed first. It looks just like the output from the Style\_Popup destination shown in Figure 1. The automatic Style\_Display output comes afterward, and is shown in Figure 3.



The screenshot shows a window titled 'PROC TEMPLATE Syntax' with a 'Style Definition' tab. The code defines a style named 'Header' with the following properties: FONT\_FACE = 'Arial, Helvetica, sans-serif', FONT\_SIZE = 4, FONT\_WEIGHT = bold, FONT\_STYLE = roman, FOREGROUND = cx0033aa, and BACKGROUND = cxb0b0b0.

Element	Proportional Fonts	Empty	Fixed Fonts
Header	Regular Strong Emphasis		Regular Strong Emphasis
Row Header	Regular Strong Emphasis		Regular Strong Emphasis
Data	Regular Strong Emphasis		Regular Strong Emphasis

This is Batch Output

Note: This is the textual content  
Warning: This is the textual content  
Error: This is the textual content

Figure 3: Automatic Style\_Display Output

The PROC TEMPLATE pop-up window for the Table style class looks different from the actual source code for the Table class (Figure 4).



**Figure 4: PROC TEMPLATE Style Definition Window**

This is the actual source code for the Table class:

```
Style Table from Output
    "Controls overall table style." ;
```

Using the Style\_Display destination makes it easy to see all of the styles that you might have in a report. If you are creating a style, you will have a good idea of all the style elements that you might want to make available. It gives you one place where you can look to see how all of the style elements work together in a report. You might discover some new style elements that can be useful, like Data\_Emphasis or Header\_Strong. The styles that SAS supplies define variations of the usual Data and Header style elements that can be used for stylized trafficlighting.

There is another tagset that generates PROC TEMPLATE code. This is a very geeky tagset, and it's not recommended as a way to create your own style. The ODS\_Style tagset creates a style sheet that is a PROC TEMPLATE definition of the style without inheritance. This can be helpful if you just want to browse through the code and see all of the style definitions in their entirety, and without inheritance. It's what you would see in all of the pop-up windows with the Style\_Popup destination or the Style\_Display destination. However, they are all in one file and you don't have to move and click all over the place.

Using the ODS\_Style tagset is simple:

```
ods tagsets.ods_style stylesheet="test.tpl";
```

You will see immediately why inheritance is both good and bad. Who wants to define the foreground and background color and font for every style element, when they are all the same? Yet, it is great to see that there are borders on the Table definition. Inheritance can be a pain, but it helps tremendously when maintaining and creating a style definition.

```
style Table/
  Frame = box
  Rules = groups
  Borderwidth = 1px
  CellSpacing = 1
  CellPadding = 7
  Background = #F0F0F0
  Foreground = #002288
  BorderColor = #000000
  ContentPosition = left
  Font = ("Arial, Helvetica, sans-serif", 3, normal normal)
  ContentScrollbar = auto
  BodyScrollbar = auto
;
```

In summary, SAS has three different destinations—Style\_Popup, Style\_Display, and ODS\_Style—that can help you understand ODS and styles. These destinations are ready-made, shipped with SAS, and very helpful when working with styles. There is more to working with styles than reverse-engineering the styles that you have; you need to

understand how styles work within the ODS framework, and how they interact with your output. Knowing these things will help you write better styles, as well as help you concentrate your efforts strategically and not waste time and effort. After all, you don't want to go to a lot of effort if all you need to do is tweak your report or add a little emphasis in your report.

### USING THE ODS HTML DESTINATION TO UNDERSTAND STYLE

This section uses the HTML destination and a custom style to show how ODS interacts with styles. `example1.sas` uses the HTML destination to create two files: a file with some PROC PRINT output and a style sheet file. The style sheet file is not necessary, but it does make it clear where the style definitions and output are. That, alone, can make it easier and quicker to find what you want.

#### example1.sas:

```
ods html file="example1.html" stylesheet="example1.css";
proc print data=sashelp.class;
run;
ods html close;
```

The style sheet file, named `example1.css`, contains all of the style definitions for every style element that is defined in the default style used by ODS. The definition for the `Table` element looks like Figure 5.

```
.Table
{
  font-family: Arial, Helvetica, sans-serif;
  font-size: small;
  font-weight: normal;
  font-style: normal;
  color: #002288;
  background-color: #F0F0F0;
  border-collapse: separate;
  border-spacing: 1;
  border-width: 1px;
  border-color: #000000;
}
```

Figure 5: Table Element Definition Style Sheet File

The content file, named `example1.html`, has only the default system title and a table, which is the output from PROC PRINT. The `Table` tag that defines the table for the PROC PRINT output looks like Figure 6.

```
<table class="Table" cellspacing="1" cellpadding="7"
  rules="groups" frame="box" border="1px" bordercolor="#000000"
  Summary="Procedure Print: Data Set SASHELP.CLASS">
```

Figure 6: Table Element Definition PROC PRINT File

Notice that the HTML code for the table that is generated by PROC PRINT has an attribute, called `class`, that has a value of `Table`. `Table` is the name of the style element, as defined in Figure 4, that it will use to define how it looks.

The output looks like Figure 6.

***The SAS System***

Obs	Name	Sex	Age	Height	Weight
1	Alfred	M	14	69.0	112.5
2	Alice	F	13	56.5	84.0
3	Barbara	F	13	65.3	98.0
4	Carol	F	14	62.8	102.5

**Figure 6: PROC PRINT Output**

If you were to examine the PROC TEMPLATE code for the style and the output in the style sheet, you would see that there is almost a one-to-one correlation between the classes that are defined in the style sheet, and the style elements that are defined in the template. SAS<sup>®</sup> 9.1 introduced the concept of an abstract style class. Any style class definitions that are abstract are left out of any style definitions that are created by ODS output; abstract style classes are there only to simplify styles by using them as parents to the style classes that ODS does use. An abstract style class is where you will find the more commonly used style definitions for fonts and colors.

To see how this works, you can create your own style and use that instead of the default. This style will have only one style element—the one used by tables. In `example2.sas`, the table border is set to a width of 1 and the color `red`. The font color is set to `blue`.

**example2.sas:**

```
proc template;
  define style styles.mystyle;
    style table /
      borderwidth = 1
      bordercolor = red
      color = blue
  ;
end;
run;

ods html file="example2.html" stylesheet="example2.css" style=mystyle;
proc print data=sashelp.class;
run;
ods html close;
```

Depending on which version of SAS you are using, the log file might contain lots of complaints about missing styles. After all, ODS expects a lot of other style classes. ODS still does a good job without them—the output is not particularly attractive, but it is perfectly viewable.

The resulting style sheet is much simpler and shorter, (part of the style sheet is shown in Figure 7), than the style sheet from the previous example, as shown in Figure 5. This style sheet is nearly empty. There are some automatically generated class definitions, but the entire style sheet is only 19 lines long.

```
.table
{
  color: #0000FF;
  border-width: 1px;
  border-color: #FF0000;
}
```

Figure 7: The Entire HTML Style Sheet

Because the actual style is very simple, so is the `Table` tag in the HTML in Figure 8.

```
<table class="table"
  border="1px" bordercolor="#FF0000"
  summary="Procedure Print: Data Set SASHELP.CLASS">
```

Figure 8: Table Element Definition

The output is much simpler, and uses red and blue, as shown in Figure 9.

## The SAS System

Obs	Name	Sex	Age	Height	Weight
1	Alfred	M	14	69.0	112.5
2	Alice	F	13	56.5	84.0
3	Barbara	F	13	65.3	98.0
4	Carol	F	14	62.8	102.5
5	Henry	M	14	63.5	102.5

Figure 9: PROC PRINT Output with a Very Simple Style

Now, take a closer look at how styles behave with the help of tagsets. A *tagset* is a collection of events. Just like style elements correspond to entities in the output, so do events in a tagset. Everything that you can create in the output has an event. There is a table event, a system\_title event, and a byline event. There is even an event that creates style class definitions. Using the previous example with a different tagset will give you somewhere to start. The `Short_Map` tagset creates XML, where the tag names in the XML are the actual event names in the tagset. Looking at the XML generated by the `Short_Map` tagset will tell a lot about what is happening when the style sheet and body files are created with ODS HTML.

example3.sas is identical to example2.sas, except for the tagset that is used.

### example3.sas:

```
proc template;
  define style styles.mystyle;
    style table /
      borderwidth = 1
      bordercolor = red
      color = blue
    ;
  end;
run;
```

```
ods tagsets.short_map file="example3_map.xml" stylesheet="example3_style_map.xml"
style=mystyle;

proc print data=sashelp.class;
run;

ods tagsets.short_map close;
```

The style sheet file is simple:

```
<stylesheet>
  <styles>
    <style_class>
      </style_class>
    </styles>
  <shortstyles/>
</stylesheet>
```

The style sheet file does not show much, but you can see that the `Table` class is written by the `style_class` event.

The body file is more complex, but some events are obvious. In particular, you can easily see where PROC PRINT starts and where the table begins with a `table` event.

```
<proc_branch name="Print" value="Print">
  <leaf name="Print" value="Data Set SASHELP.CLASS">
    <page_anchor/>
    <output name="Print">
      <table>
```

You now have enough information to create your own tagset to examine styles and their behavior. Because you started with the `Table` style and the table itself, that is what you will concentrate on. You need a tagset with two events: `style_class` and `table`. Events mostly print stuff. The `PUT` statement takes care of that. There is a special command called `putvars` that prints a whole set of variables. You can use it in this tagset to print all of the style variables. `example4.sas` is the new program.

#### example4.sas:

```
proc template;
  define style styles.mystyle;
    style table /
      borderwidth = 1
      bordercolor = red
      color = blue
    ;
  end;

  define tagset tagsets.show_style;

    define event style_class;
      put "Event: " event_name nl;
      putvars style _name_ " : " _value_ nl;
      put "=====" nl;
    end;

    define event table;
      put "Event: " event_name nl;
      putvars style _name_ " : " _value_ nl;
      put "=====" nl;
    end;
  end;
end;
```

```

        end;

run;

ods tagsets.show_style file="example4_body.txt" stylesheet="example4_style.txt"
style=mystyle;

proc print data=sashelp.class;
run;

ods tagsets.show_style close;

```

The output files created by this new program are very short, but they show some interesting behavior. The style sheet file shows all of the style information, but the table in the body file does not.

Here is the style sheet file:

```

Event: style_class
CLASS: table
BORDERWIDTH: 1px
FOREGROUND: #0000FF
BORDERCOLOR: #FF0000
FRAMEBORDER: auto
CONTENTSCROLLBAR: auto
BODYSCROLLBAR: auto
=====

```

The body file looks similar, but is shorter:

```

Event: table
CLASS: table
FRAMEBORDER: auto
CONTENTSCROLLBAR: auto
BODYSCROLLBAR: auto
=====

```

The question is, "Where is the style information in the `table` event?" It's not there. This is by design. Most markup languages have a concept of a style sheet. If all of the style information goes in the style sheet, then there is no reason to put it anywhere else. ODS automatically screens most of the style variables so that they don't show up everywhere. Only if there is a style override do the style variables begin to show up in events other than `style_class`. There is a trick you can do to see all of the style variables, but it is not usually necessary. `example5.sas` optimizes the code and uses the ODS Markup-embedded style sheet behavior to make things easier to look at. The tagset has two new attributes: `embedded_stylesheet=yes` and `pure_style=yes`. The `embedded_stylesheet` attribute tells the tagset that it is okay to put the style sheet in the body file, if no style sheet is specified. The `pure_style` attribute tells the tagset that style variables are not to be screened. At the same time, the tagset could be made simpler by creating a new event, `show_style`, that can be used to print the style information from anywhere you want.

**example5.sas:**

```

proc template;
  define style styles.mystyle;
    style table /
      borderwidth = 1
      bordercolor = red
      color = blue
    ;
  end;

  define tagset tagsets.show_style;

```



```

        embedded_stylesheet = yes;
        pure_style = yes;

        define event style_class;
            trigger show_style;
        end;

        define event table;
            trigger show_style;
        end;

        define event show_style;
            put "Event: " event_name nl;
            putvars style _name_ " : " _value_ nl;
            put "======" nl;
        end;

    end;

run;

ods tagsets.show_style file="example5.txt" style=mystyle;

proc print data=sashelp.class;
run;

ods tagsets.show_style close;

```

The output from this example is a single file, and it looks more like what you might have expected from example4.sas:

```

Event: style_class
CLASS: table
BORDERWIDTH: 1px
FOREGROUND: #0000FF
BORDERCOLOR: #FF0000
FRAMEBORDER: auto
CONTENTSCROLLBAR: auto
BODYSCROLLBAR: auto
=====
Event: table
BORDERWIDTH: 1
FOREGROUND: #0000FF
BORDERCOLOR: #FF0000
FRAMEBORDER: auto
CONTENTSCROLLBAR: auto
BODYSCROLLBAR: auto
=====

```

What do you now know from all of this? First, you know that the style information in the ODS style is used to create a style sheet by ODS Markup. You looked at HTML, but this same behavior can be seen in SpreadsheetML (which is used by Microsoft Excel), in RTF, and even in LaTeX. It is the common construct for the new office application XMLs, the Oasis Open Document Format, and Microsoft Office Open.

You also know that the style attributes are not distributed throughout the document. They are referenced back to the style sheet by name. You know that the tagset has controls to enable embedded style sheets and external style sheets, and to propagate style attributes everywhere, although it's questionable whether that is a good idea.

So far, you have looked at the styles. You have examined the behavior of styles and style sheets using the `Table` style. But, there are lots of other style classes, and there are even more tagset events. You will know this if you play around with the `Short_Map` tagset and consider all of the things that ODS Markup does when you run your various

programs. Is there an easy way to know what style classes are being used and where? Yes, there is!

### EXAMINING ODS STYLE BEHAVIOR IN MORE DETAIL

If you were curious enough to play around with the Short\_Map tagset, you already know that it works by using a default event. What that means is that a single event can stand in for all events that are not defined explicitly. The default event and mapping tagsets are your best view into the world of ODS! ODS Markup and tagsets are your glass-bottomed boat on the sea of ODS.

Starting simple, look at what style classes were used in the PROC PRINT example. It's very easy to create a tagset that will show these style classes. At first, all you want is any event that has a style, but then you realize that you don't really want anything to do with the `style_class` event. The `style_class` event creates the style sheet, and you don't want that. What you want is to identify where the styles are actually used. It all boils down to creating a tagset that prints a message for every event that has a style, except for the `style_class` event.

#### example7.sas:

```
proc template;

  define tagset tagsets.show_class;

    embedded_stylesheet = yes;
    default_event = 'show_class';

    define event show_class;
      /* quit if there isn't a style class here and */
      /* this is style_class, we don't care about the */
      /* style sheet events */
      break /if ^htmlclass | cmp(event_name, 'style_class');

      put "Event: " event_name nl;
      put "Class: " htmlclass nl;
      put "Value: " value nl;

      put "=====" nl;
    end;

  end;

run;

ods tagsets.show_class file="example7.txt";

proc print data=sashelp.class;
run;

ods tagsets.show_class close;
```

The output is interesting, but long. One thing to note is that any event containing the string `colspec` is probably not useful. Actually, any event that has no value is probably not useful. Refine the tagset and try again.

This is the new test in the tagset, as shown in `example8.sas`. It is easy to trim down the output. The following line tells the event to not produce output if there is no value, no style class, or if the event name is `style_class`.

```
break /if ^value | ^htmlclass | cmp(event_name, 'style_class');
```

The output from this tagset reflects the ODS output items and the style classes that they use, as well as the values that they display. So, you can see that `Name` is a header, and `Alfred` is data. This is useful. It's good because if what you want to see is `Alfred` or `Alice`, you can find it. But looking through all of this, you can see the number of styles is short. You can also see that this isn't exactly what you want either. Events containing the string `colspec`, or `table`

column specifications, take a bit of room, and there is no table event listed, because it has no value! Still, this information is useful because you can see the actual values of the data in the output. You can also see what the styles are for them.

Here is an excerpt:

```

Event: system_title_setup
Class: SystemTitle
Value: The SAS System
=====
Event: system_title
Class: SystemTitle
Value: The SAS System
=====
Event: proc_branch
Class: ContentProcName
Value: Print
=====
Event: leaf
Class: ContentItem
Value: Data Set SASHELP.CLASS
=====
Event: sub_colspec_header
Class: Table
Value: Obs
=====
Event: col_header_label
Class: Header
Value: Obs
=====
Event: col_header_label
Class: Header
Value: Name

=====
Event: header
Class: Header
Value: Obs
=====
Event: header
Class: Header
Value: Name
=====
Event: header
Class: Header
Value: Sex
=====
Event: header
Class: Header
Value: Age

=====
Event: header
Class: RowHeader
Value: 1
=====
Event: data
Class: Data
Value: Alfred
=====

```

```

Event: data
Class: Data
Value: M
=====
Event: data
Class: Data
Value: 14
=====
Event: data
Class: Data
Value: 69.0
=====
Event: data
Class: Data
Value: 112.5
=====
Event: header
Class: RowHeader
Value: 2
=====
Event: data
Class: Data
Value: Alice
=====
Event: data
Class: Data
Value: F
=====

```

Refine this further. How about just a list of style classes and the events they belong to? That should be a concise list. Here is a tagset that does just that. It's the same tagset from before, but instead of printing everything, it creates an associative array of events and their styles. Then, at the end, in the finish part of the doc event, it prints everything out. This isn't perfect. If you look closely, you will see that the `header` event gets a `RowHeader` style; that's not completely true. If you look above this line, you will see that the `header` event gets both `Header` and `RowHeader` styles. This tagset does not take that into account; it just gives the last style that is used for any given event. That's okay, because events mostly use the same style over and over.

#### example9.sas:

```

proc template;

  define tagset tagsets.show_class;

    embedded_stylesheet = yes;
    default_event = 'show_class';

    define event show_class;
      /* quit if there isn't a style class here and */
      /* this is style_class, we don't care about the */
      /* style sheet events */
      break /if ^value | ^htmlclass | cmp(event_name, 'style_class');
      set $events[event_name] htmlclass;
    end;

    define event doc;
      finish:
        putvars $events "Event/Style: " _name_ " / " _value_ nl;
    end;

  end;

run;

```

```
ods tagsets.show_class file="example9.txt";

proc print data=sashelp.class;
run;

ods tagsets.show_class close;
```

The output from this tagset is short. Maybe you should lift some restrictions.

```
Event/Style: system_title_setup / SystemTitle
Event/Style: system_title / SystemTitle
Event/Style: proc_branch / ContentProcName
Event/Style: leaf / ContentItem
Event/Style: sub_colspec_header / Table
Event/Style: col_header_label / Header
Event/Style: header / RowHeader
Event/Style: data / Data
```

By changing the tagset to pay attention to events without value, you still have a short list. Of these events, many of them are probably best ignored. Common sense can go a long way here. Compare this list with the previous list. If the name is obvious and makes sense, you might care about it, like the `table` event. The `cellspec` and `colspec` events are probably not so important. In reality, the list of style classes you care about is fairly short. It's actually the list above, plus `Body`, `Table`, `Byline`, `SystemTitle2-10`, and `SystemFooter-SystemFooter10`.

```
Event/Style: doc_head / Body
Event/Style: doc_meta / Body
Event/Style: auth_oper / Body
Event/Style: doc_title / Body
Event/Style: embedded_stylesheet / Body
Event/Style: styles / Body
Event/Style: javascript / Body
Event/Style: startup_function / StartUpFunction
Event/Style: shutdown_function / ShutDownFunction
Event/Style: doc_body / Body
Event/Style: anchor / Body
Event/Style: page_setup / Body
Event/Style: system_title_setup_group / Body
Event/Style: title_setup_container / SysTitleAndFooterContainer
Event/Style: title_setup_container_specs / Body
Event/Style: system_title_setup / SystemTitle
Event/Style: system_title_group / Body
Event/Style: title_container / SysTitleAndFooterContainer
Event/Style: title_container_specs / Body
Event/Style: system_title / SystemTitle
Event/Style: proc_branch / ContentProcName
Event/Style: leaf / ContentItem
Event/Style: page_anchor / PagesItem
Event/Style: table / Table
Event/Style: cellspec / Table
Event/Style: cellspecsep / Table
Event/Style: colspec_entry / Header
Event/Style: sub_colspec_header / Table
Event/Style: table_headers / Table
Event/Style: header_spec / Table
Event/Style: sub_header_colspec / Data
Event/Style: col_header_label / Header
Event/Style: row / Table
Event/Style: header / RowHeader
Event/Style: data / Data
```

At this point, you know what most of the style classes are; it's not even a very long list: `Body`, `Table`, `Header`, `RowHeader`, `Data`, `SystemTitle1-10`, and `Byline`. If you care about the table of contents, `ContentProcName`, `ContentItem`, and possibly `PagesItem` are important. This is a short list—if you have ever looked at the style definitions, “short” is not a word that comes to mind. Although, this list, plus classes for `Graph` and `Batch`, make up the short list for creating a style from scratch.

It seems like all of the styles that you get with SAS might be overkill. But, life is not always so simple. There is a need for all of those other styles. It's not frequent, but there is a need. There are errors, warnings, and notes. Sometimes, there is a page break that should be different, or a table heading that needs borders just a certain way. There is always something to make styles more complicated.

## ADDING TO STYLES AND TAGSETS FOR YOUR CUSTOM NEEDS

While it is possible that you want to create a style of your own, it is more probable that you want to do a few other things instead. You might want to modify the look of a style, its font, or table borders, and a few other things here and there. Another thing that you probably want to do is create a custom style class that you can use at will, something that highlights a value that is out of range or of more importance.

These things are easy to accomplish if you know a little bit about styles. A few examples will help get you there. Because this paper is about tagsets and styles, it is good to talk about how tagsets and styles can work together.

There are a few different scenarios where the power of tagsets and their close relationship with styles really show. In HTML tables, there is a concept of a table head, body, and foot. These are delineated by `<head> </head>`, `<body> </body>`, and `<foot> </foot>` tags. Each of these elements can have a style, but, mostly for historical reasons, these elements do not have a style in ODS output. The good thing about these HTML elements is that they can have a profound effect on how the tables look.

ODS automatically assigns particular style classes to particular sections of the output. A system title automatically gets assigned the `SystemTitle1-10` style class. So do tables, and their header and data cells. The style class names are automatically assigned for everything that ODS creates. Tagsets are more granular than that. There are many tagset events that do not have a style automatically assigned. These events get the style class that is currently active, but they are not assigned a specific style. This can be seen in the output from the previous examples. There are many events that had a style of `body` or `table`. This is not because the style was explicitly assigned, but because the event had no style of its own.

Tagset events *can* specify a style of their own! If you think about it, this might seem silly. After all, if the style is really defined in the style sheet, and only the name of a few other attributes are actually ever used, why not just code the tagset with the name. In HTML, just use `put class="mystyle"` in the tag. Done deal, no sweat, no style issues! As long as `mystyle` is defined somewhere, life is good.

If you are going for quick and painless, this might be just the thing to do. Sometimes, you want more. Maybe there are some style attributes that do special things in the tag. The attributes `cellspacing`, `cellpadding`, `frame`, and `rules` in the HTML `table` tag come to mind. Another two are `htmlstyle` and `tagattr`. These attributes are useful for various reasons. They are non-standard style attributes that were introduced by Microsoft. They are being deprecated as ODS and the rest of the world move toward version 3.0 of the Cascading Style Sheet standard. Microsoft Internet Explorer is not the dominant HTML client it once was; and, with its demise, the focus on standards and consistency has come to the forefront.

Even so, there are still things that styles can do for you, even if all you really need is the name most of the time. This paper explores some of that, but first, you should look at how a tagset can use a style class in a new way.

## ADDING BORDERS TO THE TABLE HEADER IN HTML

One of many requests is to change the way table borders look. The HTML destination is a tagset, so the opportunity to make HTML work the way you want is reasonably accessible. This example shows how to put borders above and below the header section of tables. Remember, ODS does not currently associate a style with the table head, body, or foot section. But that doesn't mean that the HTML tagset can't have a style for each of these events. All the tagset needs is to specify a style for the event. When the event occurs, the style is loaded and processed, and, at the very least, the name displays and possibly other values as well. Here is the new tagset and the style to go with it.

**example11.sas:**

```

proc template;

  define style styles.mystyle;
    parent = styles.Default;
    style table_head/
      backgroundColor = cx00CCCC
      borderTopWidth = 5
      borderTopColor = red
      borderBottomWidth = 10
      borderBottomColor = red;
  ;

end;

define tagset tagsets.myhtml;
  parent = tagsets.html4;

  define event table_head;
    style = table_head;

    start:
      put "<thead";
      trigger classalign;
      trigger style_inline;
      put ">" nl;
    finish:
      put "</thead>";
  end;

end;

run;

ods tagsets.myhtml file="example11.html" style=mystyle;

proc print data=sashelp.class;
run;

ods tagsets.mystyle close;

```

This example adds a `table_head` style class to the default style. The `table_head` class has a red top border and a red bottom border, and a background color of cyan. The background color is set using a red, green, and blue (RGB) format, with two hexadecimal values for each color. 00 is no color; FF is bright color. In this case, 00CCCC means that there is no red, but there is an equal amount of green and blue, at a bright setting of CC. Equal amounts of green and blue always make cyan. If all goes well, the table head will have a top and bottom border that is red and a background that is cyan.

The tagset portion of this example changes the `table_head` event to try to use a style of `table_head`. It prints that class and style in the `<thead>` tag. The original `table_head` event printed a `<thead>` begin and end tag. Now, it adds the class attribute so that it has a style.

The resulting HTML shows these differences in Figure 10. First, a new style is added to the style sheet.

```
.table_head
{
  background-color: #00CCCC;
  border-top-width: 5px;
  border-bottom-width: 10px;
  border-top-color: #FF0000;
  border-bottom-color: #FF0000;
  border-spacing: 1;
}
```

Figure 10: Class Definition in the Style Sheet

The <thead> tag has a class attribute that specifies table\_head as its style, as shown in Figure 11.

```
<thead class="c table_head">
<tr>
<th class="r Header" scope="col">Obs</th>
<th class="l Header" scope="col">Name</th>
<th class="l Header" scope="col">Sex</th>
<th class="r Header" scope="col">Age</th>
<th class="r Header" scope="col">Height</th>
<th class="r Header" scope="col">Weight</th>
</tr>
</thead><tbody>
```

Figure 11: <thead> Tag Uses the table\_head Class

All of that looks good, but the output is not what you would expect, as shown in Figure 12. The table borders are there, but they are very skinny and the cyan color is not present. That is because of how HTML works. The cyan color is underneath the cells, so it can't be seen. The borders are there, but they are barely visible because the default style doesn't have room for them.

<i>The SAS System</i>					
Obs	Name	Sex	Age	Height	Weight
1	Alfred	M	14	69.0	112.5

Figure 12: Barely Visible Table Head Borders

The browser gives them as much space as it can. This behavior can vary from one browser to the next. This is where things actually get difficult. It's not ODS and styles, it's where it's rendered. Whether what rendered the target is Firefox, Mozilla, Safari, Microsoft Internet Explorer, or even Microsoft Word or Adobe Acrobat, each one has its own way of doing things.



Figure 13 shows the same example, as seen in Firefox. The red doesn't show, but the cyan shows through between the cells.

**The SAS System**

Obs	Name	Sex	Age	Height	Weight
1	Alfred	M	14	69.0	112.5

Figure 13: Barely Visible Cyan in Firefox

HTML is one of the easier output types to deal with because the HTML itself can be tweaked until it does what you want. Then, you go back to the ODS style and make the changes there. The worst part is that every browser works differently.

It is possible to get closer to what you want by using a different parent style. If you change the previous example to use the `minimal` style as a parent, the output is a little bit more predictable. Using the `minimal` style as a starting place for creating a new style can be a good strategy.

The output with `minimal` style as a parent style looks like Figure 14.

**The SAS System**

Obs	Name	Sex	Age	Height	Weight
1	Alfred	M	14	69.0	112.5

Figure 14: Everything Is Visible with Minimal Style as Parent

## USING STYLE OVERRIDES

These last examples showed how you can add new style classes to a style, and create new usage of those styles within a tagset, where there was no style before. In this case, the style comes into play automatically because the tagset asks for it. Styles become a set of variable settings in the tagset for an event. Many of the attributes are screened out because they are defined initially in the style sheet. Other style attributes are not screened out. There are times when you want to give a style, or change a style attribute for the current style that the output destination is using.

The Base SAS<sup>®</sup> reporting procedures enable you to do this. The `REPORT`, `TABULATE`, `PRINT`, and `FREQ` procedures have the ability to apply style overrides to ODS output. When one of these procedures changes a value, like font or color, ODS recognizes that it is different and does not screen out the value. That means that the overridden value shows up where it wouldn't have otherwise. The next example uses a simple tagset to show the style contents for the `header` and `data` cells in the output. It is a `PROC PRINT` of `sashelp.class` that shows only `name` and `age`. `PROC PRINT` specifies a red foreground color for the header for the `name` column, and a green foreground color for the data portion of the `age` column. By default, these changes are applied on top of the default style class that is being used for `header` and `data` cells, respectively.

**example13.sas:**

```

proc template;

  define tagset tagsets.show_style;

    embedded_stylesheet = yes;

    define event header;
      trigger show_style;
    end;

    define event data;
      trigger show_style;
    end;

    define event show_style;
      put "Event: " event_name nl;
      put "Value: " value nl;
      putvars style _name_ " : " _value_ nl;
      put "=====" nl;
    end;

  end;

run;

options obs=1;

ods tagsets.show_style file="example13.txt";
ods html file="example13.html";

proc print data=sashelp.class noobs;
  var name /style (header) = [foreground=red];
  var age  /style (data) = [foreground=green];
run;

ods _all_ close;

```

To make this a little easier to understand, this test creates an HTML file that shows just what this PROC PRINT has done with the style overrides, as shown in Figure 15.

Name	Age
Alfred	14

**Figure 15: PROC PRINT Output with Style Overrides**

The key part of the HTML looks like the following (it's easy to see that HTML added a style attribute with a color):

```
<th class="l Header" style=" color: #FF0000;" scope="col">Name</th>
<th class="r Header" scope="col">Age</th>
</tr>
</thead>
<tbody>
<tr>
<td class="l Data">Alfred</td>
<td class="r Data" style=" color: #008000;">14</td>
```

The custom Show\_Style tagset prints an entry for each `header` and `data` event that it receives. The HTML clearly shows that there are two headers, `Name` and `Age`, followed by two data cells for `Alfred` and his age. The entry created by the tagset shows the tagset event name, the value being passed to the event (which is the actual header or data value), as well as the style class name that is being used. Below that, all of the style attributes that are allowed past the screening process are shown. What you see is that the header for `Name` has a foreground color of `#FF0000`, which is red. The second header, `Age`, is untouched. It does have `frameborder`, `contentscrollbar`, and `bodyscrollbar` attributes, but all of the style classes show those attributes because of the way the default style is written.

The next two entries are for the two data cells. The first data cell is `Name`, and that style is untouched from the definition. The second data cell is `Age`, and it is easy to see that the foreground color is being set to green.

```
Event: header
Value: Name
CLASS: Header

FOREGROUND: #FF0000
FRAMEBORDER: auto
CONTENTSCROLLBAR: auto
BODYSCROLLBAR: auto
=====
Event: header
Value: Age
CLASS: Header
FRAMEBORDER: auto
CONTENTSCROLLBAR: auto
BODYSCROLLBAR: auto
=====
Event: data
Value: Alfred
CLASS: Data
FRAMEBORDER: auto
CONTENTSCROLLBAR: auto
BODYSCROLLBAR: auto
=====
Event: data
Value: 14
CLASS: Data
FOREGROUND: #008000
FRAMEBORDER: auto
CONTENTSCROLLBAR: auto
BODYSCROLLBAR: auto
=====
```

The important thing to see is how this all works. It's just information that is being passed to the destination by the procedure. When it comes to the tagset destinations, this means that you can use styles as a mechanism for controls, even beyond style.

Another thing to think about is the readability of your code. Lots of style attribute overrides can cause the procedure code to become messy and harder to read. It creates processing that might happen thousands of times in a procedure run. Instead of doing style-attribute-level overrides, you can create a style that has definitions for the changes you want.

Adding a style to the previous example shows the effect on the procedure code and in the behavior of the output. The addition of a style might seem to complicate things, until you want to use the exact same settings in another procedure or program. Then, you will be happy that the style is there and it has all of your settings, just the way you want them. The procedure code is simpler and easier to read. This is more apparent as the style changes become more complex.

**example14.sas:**

```
proc template;
  define style styles.mystyle;

    style red_header from header/
      foreground=red
    ;

    style green_data from data/
      foreground=green
    ;

  end;

  define tagset tagsets.show_style;

    embedded_stylesheet = yes;

    define event header;
      trigger show_style;
    end;

    define event data;
      trigger show_style;
    end;

    define event show_style;
      put "Event: " event_name nl;
      put "Value: " value nl;
      putvars style _name_ " : " _value_ nl;
      put "=====" nl;
    end;

  end;

run;

options obs=1;

ods tagsets.show_style file="example14.txt" style=mystyle;
ods html file="example14.html" style=mystyle;

proc print data=sashelp.class noobs;
  var name /style (header) = red_header;
  var age /style (data) = green_data;
run;

ods _all_ close;
```

The output for both the HTML and the Show\_Style tagset is simpler, although the HTML looks exactly the same as before. The foreground color is no longer present for either case, although the style class names change just the way that you asked them to.

```
<th class="l red_header" scope="col">Name</th>
<th class="r Header" scope="col">Age</th>
</tr>
</thead>
<tbody>
<tr>
<td class="l Data">Alfred</td>
<td class="r green_data">14</td>
```

All that changes in the Show\_Style output is the style class name.

```
Event: header
Value: Name
CLASS: red_header
FRAMEBORDER: auto
CONTENTSCROLLBAR: auto
BODYSCROLLBAR: auto
=====
Event: header
Value: Age
CLASS: Header
FRAMEBORDER: auto
CONTENTSCROLLBAR: auto
BODYSCROLLBAR: auto
=====
Event: data
Value: Alfred
CLASS: Data
FRAMEBORDER: auto
CONTENTSCROLLBAR: auto
BODYSCROLLBAR: auto
=====
Event: data
Value: 14
CLASS: green_data
FRAMEBORDER: auto
CONTENTSCROLLBAR: auto
BODYSCROLLBAR: auto
=====
```

Usually, the distinction in behavior is not so important. The processing doesn't really matter. The big benefits are reusability of your style tweaks, and readability of your procedure code. There is one case where the behavior distinction matters greatly—that is when you are using the tagsets.excelxp destination. The processing time and XML that is generated by the tagset can be significantly increased if there are style attribute overrides at the procedure level. Unlike HTML, where the style overrides can be placed directly around the data, the Microsoft Excel XML must define a style class, much like in the HTML style sheet. The ExcelXP tagset must process these overrides and create new style entries. Sorting all of that out is expensive in processing and in the size of the XML file that is generated. Creating and using a style definition like the last example avoids all of that.

## THE EXCELXP DESTINATION AND STYLE OVERRIDES

From all of this behavior, it is clear that styles can provide programmatic changes in various attributes that are available in the output destination. Because ODS Markup is also programmable, using styles in this way can cause changes in the behavior of the ODS Markup destinations. The most obvious change in behavior can be seen in the ExcelXP tagset. In this tagset, there are several behavior changes that can occur because of settings in the styles. The ExcelXP tagset looks for formats, formulas, and text rotation in a mostly unused style attribute called `tagattr`. The `tagattr` attribute is not screened out by the style sheet, so it is always available, even when it is used in a style

definition similar to the last example. The `tagattr` attribute escapes any processing by ODS, so any free-form text can be put there, and that's what the destination sees when the style comes out of ODS.

The ExcelXP tagset looks for other attributes. For example, `cellwidth` can control column widths in a worksheet with absolute control. The `flyover` attribute can be used to add comments to worksheet cells.

There are a few style classes that are automatically generated if they do not exist. For example, `datamissing` changes the way missing values are rendered. The `parskip` class can be used to control the amount of blank space in empty worksheet rows.

This next example shows how to use the `tagattr` attribute to control the look of a worksheet. The ExcelXP tagset looks for four different settings in the `tagattr` attribute: `rotate`, `type`, `format`, and `formula`. This example uses `rotate` in a style element to change the orientation of the headers.

The style part of this is easy; there is one new style class called `angle_header`. The value of `tagattr` is `rotate:45`. This syntax is specific to the ExcelXP tagset and can be seen in the tagset documentation by issuing this ODS statement:

```
ods tagsets.excelxp options(doc='help');
```

The `tabulate` portion of this program is nothing special. The syntax for assigning a style class is not unlike the syntax for PROC PRINT. In this case, the `angle_header` style class is assigned to the `classlev` variable with the terse `tabulate` syntax of `/s=angle_header`.

#### example15.sas:

```
proc template;
  define style styles.myjournal;
    parent=styles.Journal;

    style angle_header from header /
      tagattr='rotate:45'
    ;
  end;
run;

ods tagsets.ExcelXP file="Example15.xls" style=myjournal;

proc sort data=sashelp.class out=foo;
  by sex;

proc tabulate data=foo
  order=data missing format=8.0 noseps
  formchar=',          ';
  by sex;

  class age sex name;

  classlev name /s=angle_header;

  var height weight;
  title;
  table age, name=' *(height=' '*median=' '*F=5.3);

run;

ods tagsets.ExcelXP close;
```

As seen in Figure 16, the output from this program looks very good, considering how little work went into it.

	Alice	Barbara	Carol	Jane	Janet	Joyce	Judy	Louise	Mary
Age	56.5	65.3							
13			62.8				64.3		
14				59.8				56.3	
12					62.5				66.5
15						51.3			
17									

Figure 16: ExcelXP Output Created with Style Overrides That Rotate the Header

This next example shows formats and formulas when they are defined in a style and when they are used as a style override. Either way works perfectly fine—it's for you to decide what's best for you. In this example, the last three columns of the worksheet have the same formats, where negative values are displayed in the color red and in parentheses. The last column also has a formula, so it will be recalculated when values are changed in the two preceding columns.

First, define a style. The first PROC PRINT does not use the new style classes, but the second PROC PRINT does. The output for both PRINT procedures looks the same.

```
proc template;
  define style styles.mystatistical;
    parent=styles.statistical;

    style data_formatted from data /
      tagattr='format:$#,##0_);[Red]\($#,##0\)'
    ;

    style data_difference from data /
      tagattr='format:$#,##0_);[Red]\($#,##0\)' formula:RC[-1]-RC[-2]'
    ;
  end;
run;

/*-----*/
/*-- Create data --*/
/*-----*/
data prdsale;
  set sashelp.prdsale;
  Difference = actual-predict;
run;

proc sort data=prdsale;
  by country region division year;
run;

ods tagsets.ExcelXP file="example16.xls" style=mystatistical
  options(autofilter='1-3'
          frozen_headers='2'
          frozen_rowheaders='4'
          auto_subtotals='yes');
```

```

/*-- Use Excel formulas to represent computed cells, --*/
/*-- and use an Excel format to force Excel to show --*/
/*-- negative currency values in red and with the --*/
/*-- format ($nnn). In the formula below, the RC --*/
/*-- value corresponds to the cell relative to the --*/
/*-- current cell. For example, RC[-2] means "2 --*/
/*-- cells to the left of the current cell". Any --*/
/*-- valid Excel formula can be used, and the formula --*/
/*-- used here matches the computation performed --*/
/*-- in the DATA step that created the column. --*/
/*-----*/
proc print data=prdsale noobs label split='*';
  id country region division;
  var prodtype;
  var predict actual / style={tagattr='format:$#,##0_);[Red]\($#,##0\)'};
  var difference / style={tagattr='format:$#,##0_);[Red]\($#,##0\)
    formula:RC[-1]-RC[-2]'};
  sum predict actual difference /

style={tagattr='format:$#,##0_);[Red]\($#,##0\)'};
  label prodtype = 'Product*Type'
    predict = 'Predicted*Sales*For Area'
    actual = 'Actual*Sales*Amount';
run;

proc print data=prdsale noobs label split='*';
  id country region division;
  /*var prodtype product quarter month year;*/
  var prodtype;
  var predict actual / style(data)=data_formatted;
  var difference / style(data)=data_difference;
  sum predict actual difference / style(data)=data_formatted;;
  label prodtype = 'Product*Type'
    predict = 'Predicted*Sales*For Area'
    actual = 'Actual*Sales*Amount';
run;

ods tagsets.ExcelXP close;

```

The output looks like Figure 17.

	A	B	C	D	E	F	G	H
1	Country	Region	Division	Product Type	Predicted Sales For Area	Actual Sales Amount	Difference	
472	CANADA	WEST	EDUCATION	OFFICE	\$603	\$187	(\$416)	
473	CANADA	WEST	EDUCATION	OFFICE	\$506	\$415	(\$91)	
474	CANADA	WEST	EDUCATION	OFFICE	\$185	\$61	(\$124)	
475	CANADA	WEST	EDUCATION	OFFICE	\$692	\$839	\$147	
476	CANADA	WEST	EDUCATION	OFFICE	\$565	\$596	\$31	
477	CANADA	WEST	EDUCATION	OFFICE	\$512	\$751	\$239	
478	CANADA	WEST	EDUCATION	OFFICE	\$86	\$460	\$374	
479	CANADA	WEST	EDUCATION	OFFICE	\$399	\$922	\$523	
480	CANADA	WEST	EDUCATION	OFFICE	\$672	\$153	(\$519)	
481	CANADA	WEST	EDUCATION	OFFICE	\$801	\$928	\$127	
1443								
1444								

Figure 17: ExcelXP Output with Formats and Formulas



## USING STYLES TO CONTROL A CUSTOM HTML TAGSET

This last example uses the HTML tagset. This tagset does something special—it looks for particular styles. If it finds them, then it behaves one way; if it doesn't find them, then it behaves another way. This behavior enables the tagset to create reasonable output with limited resources, and to create great output if it is given more resources.

The HTML tagset creates slide bars in its data cells, but only in the data cells that you ask it to create slide bars in. If you create a good style definition and image for the HTML tagset, it can generate very attractive slide bars. If you don't, it still makes good slide bars.

The key to all of this information is using a combination of style attributes. You are already familiar with `tagattr`, but there are other attributes that can be used in similar ways. There are a set of attributes that normally put images, text, or HTML before or after any item that they are set on, including `preimage` and `postimage`, `pretext` and `posttext`, and `prehtml` and `posthtml`. None of these attributes is used in the style sheet definitions, and all of these attributes escape screening by the style sheet behavior of ODS.

This HTML tagset uses `tagattr` to turn on the slide bar; if there is a `preimage` attribute set in the style, that image is used to create the actual slide bar. If there is not, the tagset uses the background color of the style to create the slide bar. For that reason, you want to create or use a style class that contrasts against the data style. One simple and reliable approach is to use the `header` style. In this example, you create a simple `slider` style from the `data` style with a different background color.

The `tagattr` attribute is used to turn on the slide bar and `preimage` is used to create the fancy slide bar. If `preimage` is not there, then the tagset does without an image to create the slide bar. The decisions are all made in the tagset's data event.

Here is what the tagset does. First, it looks for the word "slider" in `tagattr`. Then, it calculates the width of the slider. For simplicity, it sets a variable, named `slider_column`, to `True`. That makes life easier because the tagset needs to know what mode it is in. Next, if there is a `preimage`, it sets `bar_column` to `True`. It creates the `data_cell_value`, and it's done. If there is no `preimage`, it has a lot more work to do. It creates a cell and table with the `data` style, and then goes on from there. The next steps are to create two cells in that table: one cell with the contrasting background color and one cell without it.

```
do /if index(tagattr, 'slider') > 0;
  trigger calculate_width;
  set $slider_column "True";
  do /if preimage;
    set $bar_column "True";
    trigger data_cell_value;
    break;
  else;
    put '<td width="150" class="data">' nl;
    put '<table width="100%" class="data" cellpadding=0; cellspacing=0;>' nl;
  done;
done;
trigger data_cell_value;
```

The style portion of this example is not much different from the other examples. You are just adding some new class definitions to an existing style. The first style is for the simple slider, and the second style is for the fancy image-based slider.

```
style slider from data /
  background = colors('headerbg')
;

style bar from data /
backgroundimage = 'gridline58.gif'
borderleftwidth = 1px
borderleftstyle = solid
borderleftcolor = cxe5e5e5
borderrightwidth = 1px
```

```

borderrightstyle = solid
borderrightcolor = cxe5e5e5
cellpadding = 0
borderbottomstyle = none
preimage = 'bar.png';
;

```

There is one more thing to think about. What value will the width of the slide bar be calculated from? Will it be a percentage? That would be ideal. But, what if it just a number? The tagset needs to know how to create a percentage so that it can create the properly sized slide bar. That value can be passed with `tagattr`. You put the maximum value for that field in `tagattr`, right after the keyword `slider`. Using `sashelp.class` means that you need to supply maximum values for `height` and `weight` if you want a slide bar to show up. If all you want is the simple slide bar, use the `slider` style class.

```

proc print data=sashelp.class;
  var name;
  var sex;
  var age;
  var height / style(data) = slider[just=center tagattr="slider-80"];
  var weight / style(data) = slider[just=center tagattr="slider-150"];
run;

```

If you want the fancy slide bar, use the `bar` style class. Other than that, the code is nearly the same.

```

proc print data=sashelp.class;
  var name;
  var sex;
  var age;
  var height / style(data) = bar[just=left
tagattr="slider-80"];
  var weight / style(data) = bar[just=left
tagattr="slider-150"];
run;

```

The output from the first PROC PRINT uses nested tables and cells to create the slide bars, and it looks like Figure 14.

**The SAS System**

Obs	Name	Sex	Age	Height	Weight
1	Alfred	M	14	69.0	112.5
2	Alice	F	13	56.5	84.0
3	Barbara	F	13	65.3	98.0
4	Carol	F	14	62.8	102.5
5	Henry	M	14	63.5	102.5
6	James	M	12	57.3	83.0
7	Jane	F	12	59.8	84.5
8	Janet	F	15	62.5	112.5
9	Jeffrey	M	13	62.5	84.0
10	John	M	12	59.0	99.5
11	Joyce	F	11	51.3	50.5
12	Judy	F	14	64.3	90.0
13	Louise	F	12	56.3	77.0
14	Mary	F	15	66.5	112.0
15	Philip	M	16	72.0	150.0
16	Robert	M	12	64.8	128.0
17	Ronald	M	15	67.0	133.0
18	Thomas	M	11	57.5	85.0
19	William	M	15	66.5	112.0

**Figure 18: Slide Bars without Images**

As shown in Figure 19, the output from the second PROC PRINT uses a stretched image to create slide bars, and it looks fancier than the other method. Both look good, considering how few additions you made to the original style.

**The SAS System**

Obs	Name	Sex	Age	Height	Weight
1	Alfred	M	14	69.0	112.5
2	Alice	F	13	56.5	84.0
3	Barbara	F	13	65.3	98.0
4	Carol	F	14	62.8	102.5
5	Henry	M	14	63.5	102.5
6	James	M	12	57.3	83.0
7	Jane	F	12	59.8	84.5
8	Janet	F	15	62.5	112.5
9	Jeffrey	M	13	62.5	84.0
10	John	M	12	59.0	99.5
11	Joyce	F	11	51.3	50.5
12	Judy	F	14	64.3	90.0
13	Louise	F	12	56.3	77.0
14	Mary	F	15	66.5	112.0
15	Philip	M	16	72.0	150.0
16	Robert	M	12	64.8	128.0
17	Ronald	M	15	67.0	133.0
18	Thomas	M	11	57.5	85.0
19	William	M	15	66.5	112.0

**Figure 19: Slide Bars with Images**

## CONCLUSION

You have come a long way in this paper. You first looked at the available styles, and you examined various ways to look at those styles. Through conventional methods and not-so-conventional methods using tagsets, I have shown you how tagsets can be used to explore styles, both their behavior and their contents. Tagsets can show you how styles behave within ODS, and with ODS Markup. You have seen how styles can be used by procedures to change the way output looks. You have seen how styles and procedures can work together.

Finally, by using the programmability of tagsets, I have shown you how styles can be used to not only change how your ODS output looks, but also to change the way tagsets behave. This ultimately results in even more dramatic changes in the way output looks.

All of the examples in this paper have used simple style definitions that are common to everyday work. These small additions and changes to styles can make a world of difference in both the way your SAS output looks, and the effort that is required to create it. Styles can be a huge asset in saving time and creating beautiful output for your customers.

## CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the author:

Eric Gebhart  
 SAS Institute Inc.  
 E-mail: [Eric.Gebhart@sas.com](mailto:Eric.Gebhart@sas.com)  
 Web: [www.sas.com](http://www.sas.com)

SAS and all other SAS Institute Inc. product or service names are registered trademarks or trademarks of SAS Institute Inc. in the USA and other countries. ® indicates USA registration.

Other brand and product names are trademarks of their respective companies.