Paper 3-27

MARKUP: The Power of Choice and Change.

Eric Gebhart, SAS® Institute, Cary, NC

ABSTRACT

When it comes to markup languages, there are lots of choices. If you are keeping up with the web, you might say XML is your choice. XML alone is full of choices. But XML is just the latest craze. There are lots of other markup languages out there. Using the ODS MARKUP output destination the ability to create these types of output are within your grasp.

INTRODUCTION

Everyone wants your data their way. Perhaps your choice is XHTML, WML, CHTML, I-Mode, or just plain old HTML. The web is a fast-paced place! But don't forget there are other markup languages. How about CSV (Comma Separated Values), or LaTeX, or even Troff, and what about those scripts you run that rip the listing output to pieces. Wouldn't it be nice if you could customize your output to be exactly the way you wanted? With the ODS MARKUP destination that power is yours. Through the power of ODS MARKUP tagsets you can specify every detail about your output. New flavors of XML, new markup languages, or changes to the smallest of details are now within your grasp. This paper will talk about using ODS MARKUP to create and modify ODS output destinations for every need.

Reasonable people adapt themselves to the world. Unreasonable people attempt to adapt the world to themselves. All progress, therefore, depends on unreasonable people." George Bernard Shaw

THE WEB IS UNREASONABLE

The Web is unreasonable because it wants us to conform to it. Unfortunately the web is also a living thing. It changes as it grows. The web evolves very quickly. The important thing to note is that it's growth is driven by unreasonable people. We have a choice. We can conform or we can be unreasonable. Neither side of the fence is easy. To conform we have to keep up with the evolution. To be unreasonable not only do we have to keep up, we have to be knowledgeable, informed, quick to implement and above all, imaginative.

THE POWER TO CHOOSE

It is not my place to choose your place on the wave of evolution. But until now, I have. You have had to live with the markup output that I decided you should have. The end result is full of compromises and consolations. But now I am stepping aside. I am giving you the power to choose your place on the wave. Even if all you are trying to do is keep up with all those unreasonable people out there.

THE POWER TO BE UNREASONABLE

Because I am unreasonable I am giving you the power to be unreasonable with me. Whether you are just trying to keep up, or to shape the world, I want you to have choices. Even in keeping up you can be just a little unreasonable. This is how the web grows and evolves. Even your client relations evolve this way. You now have the power to conform or be unreasonable if you so choose.

"Choice has always been a privilege of those who could afford to pay for it." Ellen Frankfort

THE XML FLAVOR OF THE WEEK

XML alone is full of choices. Everyone seems to have an emerging standard. Everyone wants your data their way. Just for generic use there is OASIS' Docbook, there's OIM, Biztalk, and more. Not to mention all the industry specific XML definitions.

YOU SAY POTATO, I SAY ....

Everyone seems to have an opinion about how they like their HTML. Some want it pretty, some want it small, some like tables for formatting, some don't. Frames, No Frames, stylesheets, No stylesheets, style classes, styles by tag. From my point of view HTML is more than high maintenance, HTML is a bottomless pit. I'm calling the whole thing off. From now on it's your choice not mine. But there is a price. It's knowledge.

KNOWLEDGE IS POWER. - Francis Bacon

But first let's get dangerous. Here are some simple real world examples to get us started. In the current ODS html output we create anchors with a non breaking space in them. They look something like this.

```
<a name=IDX>&nbsp;</a>
```

Lots of people don't notice. Other people hate the space created by the non-breaking space. Using tagsets it is possible to change the anchor to anything you want, or even remove it. This is what you would do.

```
proc template;
  define tagset tagsets.myhtml;
    parent=tagsets.htmlcss;
    define event anchor;
      putq "<a name=" NAME ">" NL /
            exists(NAME);
    end;
  end;
run;
```

You only have to run the above code once. From then on your new tagset will be available for you to use whenever you want. Then to use your new tagset just use an ODS statement similar to this.

```
ods tagsets.myhtml file="body.html"
stylesheet="style.css";
...
ods tagsets.myhtml close;
```

You must be the change you wish to see in the world. *Gandhi*

CHANGING A TAGSET THROUGH INHERITANCE.

Tagsets can inherit from each other. Inheritance is great for tweaking a tagset to your needs without rewriting the whole thing. Many of the supplied tagsets are created this way. Just add a line like the one below to the top of your tagset definition.

```
Parent = tagsets.Event_map
```

This type of change is the easiest and most common. You already know which tagset you are using. You've looked at the output and know what you want to change. The next thing is to find out how to change it. You can do that a couple of ways. You can look at the tagset within SAS. In this case we want to look at the htmlcss tagset. This is how to do that.

THE STANDARDS

The default tagsets are kept in the Master template in the Tagsets directory.
You can see them by doing this:

```
Proc template;
list tagsets;
```

As of this writing the output from this statement is this:

```
Listing of: SASHELP.TMPLMST
Path Filter is: Tagsets
Sort by: PATH/ASCENDING

Obs    Path                          Type
 1     Tagsets                       Dir
 2     Tagsets.Chtml                 Tagset
 3     Tagsets.Colorlatex            Tagset
 4     Tagsets.Csv                   Tagset
 5     Tagsets.Csvall                Tagset
 6     Tagsets.Csvbyline             Tagset
 7     Tagsets.Default               Tagset
 8     Tagsets.Docbook               Tagset
 9     Tagsets.Event_map             Tagset
10     Tagsets.GTableApplet          Tagset
11     Tagsets.Graph                 Tagset
12     Tagsets.Html4                 Tagset
13     Tagsets.Htmlcss               Tagset
14     Tagsets.Imode                 Tagset
15     Tagsets.Latex                 Tagset
16     Tagsets.Latex2                Tagset
17     Tagsets.Namedhtml             Tagset
18     Tagsets.ODSstyle              Tagset
19     Tagsets.Phtml                 Tagset
20     Tagsets.Pyx                   Tagset
21     Tagsets.SASReport             Tagset
22     Tagsets.SASxmog               Tagset
23     Tagsets.SASxmoh               Tagset
24     Tagsets.SASxmoim              Tagset
25     Tagsets.SASxmor               Tagset
26     Tagsets.Short_map             Tagset
27     Tagsets.Statgraph             Tagset
28     Tagsets.Style_display         Tagset
29     Tagsets.Style_popup           Tagset
30     Tagsets.Text_map              Tagset
31     Tagsets.Tpl_style_list        Tagset
32     Tagsets.Tpl_style_map         Tagset
33     Tagsets.Troff                 Tagset
34     Tagsets.Wml                   Tagset
35     Tagsets.Wmlolist              Tagset
36     Tagsets.SASXML                Tagset
37     Tagsets.SASioXML              Tagset
```

You can see the source for any of the tagsets. Here is how to you can see the source of the csv tagset.

```
Proc template;
source tagsets.csv;

NOTE: Path 'Tagsets.Csv' is in:
SASHELP.TMPLMST.
define tagset Tagsets.Csv;
   notes "This is the CSV definition";
   define event table;
      start:
         put NL;
      finish:
         put NL NL;
   end;
   define event row;
      put NL;
   end;
   define event header;
      start:
         put "," / if !cmp( COLSTART , "1" );
         put """";
         put VALUE;
      finish:
         put """";
   end;
   define event data;
      start:
         put "," / if !cmp( COLSTART , "1" );
         put """";
         put VALUE;
      finish:
         put """";
   end;
   define event colspanfill;
      put ",";
   end;
   define event rowspanfill;
      put ",";
   end;
   define event breakline;
      put NL;
   end;
   define event splitline;
      put NL;
   end;
   output_type = "csv";
   mapsub = "/""""/(c)/(r)/(TM)/";
   map = """□□□";
   stacked_columns = OFF;
end;
```

Now it's just a matter of searching for the thing you want to change and redefining the event it belongs to. The second way to look at the tagset is to download all the tagsets from this link http://www.SAS.com/rnd/base/index-early-access.html and look at them in your favorite editor.

A common complaint about the csv tagset is that there are two blank lines at the beginning of the file. Looking at the above code we can see that both the table and row events start by creating a new line, NL. We can fix this by crating a new tagset with the following code.

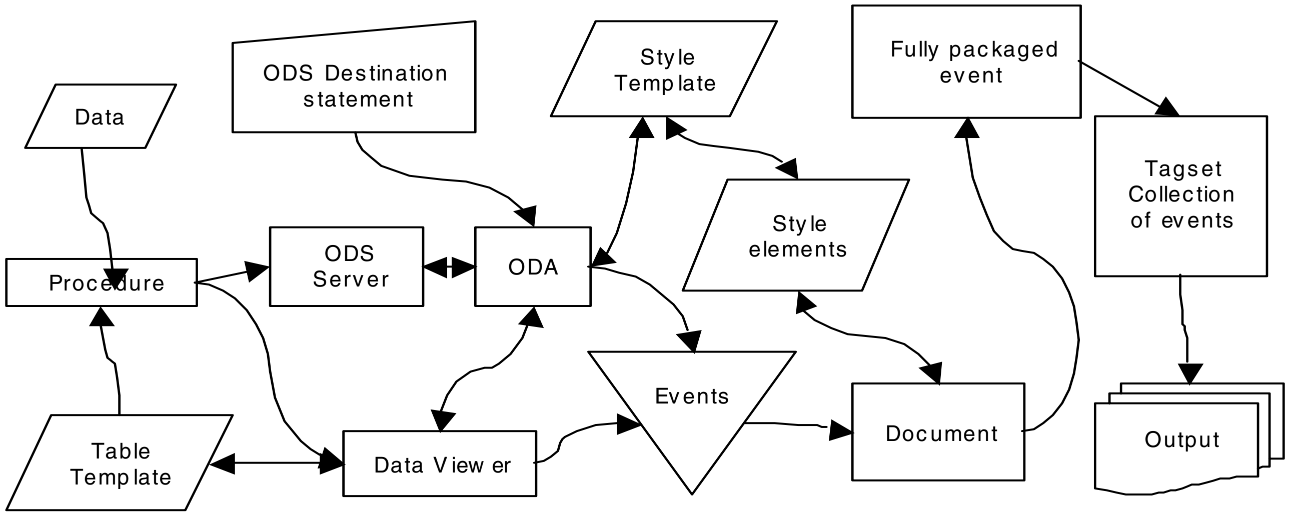
```
Proc template;
   define tagset Tagsets.NewCsv;
      parent=tagsets.csv;

      define event table;
        finish:
           put NL;
      end;

      define event row;
        finish:
          put NL;
      end;
   end;
run;
```

There are no mistakes, no coincidences. All events are blessings given to us to learn from. *Elizabeth Kubler-Ross*

So what are these events anyway? Events are places in time. Events begin and end. Events can occur between the beginning and ending of other events.



Events ultimately come from the SAS job you are running. They come from the ODS statement you used to invoke ODS MARKUP. They come from ODS MARKUP's Output Delivery Agent or ODA. The ODA supplies the outlying structure to all of your output. It does things like page breaks and system titles; It also creates the hierarchical table of contents. MARKUP's ODA is who you are talking to when you issue the ODS MARKUP statement. Think of the ODA as the all-knowing overseer of your output.
But there are all the data tables, notes and bylines. Those ultimately come from the proc you are running. But the procedure is really just talking to the ODA, and telling it what it wants. When it has data to be viewed it tells the ODA. The ODA puts the procedure in touch with a viewer and then steps out of the way. The viewers know how to put the data and the formatting together to create nicely formatted output. These viewers always follow the same structure for the event's they request. For instance, the table viewer always gives us all the events which we need to create a table in all of it's various forms. But there are a lot of possible events the viewer needs to request. I say request, because that is really what happens. All events come to the tagset as a request. At this point the event is carrying the data, the formatting attributes from the table template, and the attributes from the assigned style element. If the tagset defines that event then something will happen, output will be generated. If the tagset doesn't define that event then nothing happens. Any normal tagset will never define all the events that are possible. The reason for that is because the set of possible events generated by SAS includes what is necessary to create output for many different markup languages. Troff tables look nothing like html tables, which look nothing, like csv tables.

You might be guessing that I'm not going to tell you all the names of all the events and when they are going to occur. You would be right. But don't panic. Once you get to know tagsets you'll realize that a list isn't going to help you anyway. There are better ways.

THE DEFAULT EVENT

Normally nothing will happen if ODS asks for an event which the current tagset does not define. But we have something called the 'default event'. We just tell the tagset which event we want it to use and suddenly we have the largest most verbose output you can imagine. But this can be incredibly useful. If you really want that definitive list of event names you could get it by creating a tagset like this one.

```
proc template;
  define tagset tagsets.eventnames;
    default_event='doit';
    define event doit;
      put event_name nl;
    end;
  end;
run;
```

Of course now you have to run some jobs through it, and then sort the list and get rid of the duplicates. But If you really wanted that list, there you have it.

Following the light of the sun, we left the Old World. *Inscription on Columbus' caravel.*

THE MAPPING TAGSETS

There are several tagsets that put a default event to good use. The original is event_map. Event_map will generate xml that uses the event names as the tag names. It also displays a fair number of attributes and their values. Event_map is quite verbose. It's so verbose I created another tagset called short_map. It's just like event_map but only displays a handful of attributes on each event. Variations on event_map are text_map and tpl_style_map. There is a variation on tpl_style_map called tpl_style_list. It creates a nice HTML bulleted list of all the events that fire. Text_map is a very nice tagset which is more human readable that most of the other mapping tagsets. It also comes with comments that may help you if these event things still have you confused.

PYX
Pyx is another very useful tagset. Pyx is interesting because every event name and attribute is on a line by itself. It makes for a long but narrow file that is easily manipulated. It also has a number of tools written in various languages that can manipulate it, validate it and convert it to and from xml. For more information on pyx take a look at the book 'XML processing with Python' By Sean McGrath. There is also a website devoted to pyx. http://www.pyxie.sourceforge.net

Words are, of course, the most powerful drug used by mankind.

Rudyard Kipling

THE PUT STATEMENT
Creating textual output of some sort is the goal of all tagsets. To that end there is the put statement and it's variations. All variants of put take a list of strings and variables that are to be output.
Put is the basic statement in it's simplest form.
Putl automatically adds a new line to the end. This is great for when an event's output resembles a novel.
Putq places quotes around all variable values.

```
putq "color=" foreground;
```

Will result in output similar to this:

```
color="blue"
```

BREAKING LINES, CR, NL, LF
In addition to strings and variables you may also specify NL, CR or LF. All of these are treated the same. They all cause a new line to occur.

THE GEMINI.
Put statements also pair strings with variables. If a string is followed by a variable, they become a pair. If the variable has a value then the pair becomes output. If not then neither of them will be output. This is great for XML and HTML. Consider the following.

```
putq "<Table" " background" background "
      foreground=" foreground
      "cellpadding=" cellpadding ">" nl;
```

If none of the variables have a value the output would be

```
<Table>
```

The problem with variables is that they are so variable.

Tagsets have a lot of automatic variables that hold data that you can use. All of the variables from the ODS styles are there, All 69 of them. Then there are the variables that hold the basic data for the event. As of this writing there are 102 of those.
There is also another kind of variable. Dynamic variables. If you are familiar with proc template you may already be familiar with dynamic variables. Dynamic variables can be listed with a dynamic statement. Or they can be referenced by preceding them with an @. These variables are dynamic in that they are not defined within the confines of ODS. But are defined by our internal users. The biggest creator of dynamic variables is SAS Graph. Finally there are user-defined variables. They aren't anything fancy. You can assign a string or the value of another variable to them. You can also clear them. These variables are designated with a $.
Obviously not all these variables will have a value all of the time. But many of them will always be there. Many more will almost always be there. Many of the style variables will only be present if the value is an over-ride passed in on the fly. ODS MARKUP uses the concept of stylesheets. The stylesheet events will have fully populated values. But all other events will only get the values that are not in the stylesheet. The variables that are allowed to leak through are those variables that are not valid in Cascading Style Sheets, CSS. There are some more advanced tactics that allow you to over-ride the stylesheet related behavior.
Here is a sample SAS job that shows the simple relationship between ODS styles and tagsets. Notice that we don't even have to run a proc to see the output from our simple little tagset.

```
proc template;
  define Style styles.mine;
    parent =   styles.minimal;
    replace Document from Document/
      dropshadow = yes
      image = "thisimage.gif"
      watermark = yes
      transparency=20
  ;
end;
run;

proc template;
  define tagset tagsets.foo;
    define event doc;
      start:
        put "<" EVENT_NAME ;
        putq " dropshadow=" dropshadow;
        putq " transparency=" transparency;
        putq " watermark=" watermark;
        putq " asis=" asis;
        putq " tagattr=" tagattr;
        putq " image=" image;
        put ">" NL;
      finish:
        put "</" EVENT_NAME '>';
     end;
   end;
run;

ods tagsets.foo body="b.xml"
    STYLE=mine;
ods tagsets.foo close;
```

The result of this job is b.xml, which will contain the following.

```
<doc dropshadow="yes" transparency="20"
watermark="yes" image="thisimage.gif">
 </doc>
```

THE ANATOMY OF AN EVENT.
We've already discussed the put statements. But start: and finish: are just begging for explanation.
Events can be stateless. These are the simple events. They don't begin or end, they just are. They don't own or contain other events. But some events contain other events. They have a beginning, middle, and an end. Take a table for instance. It starts. Then there are a bunch of rows and cells, and then it ends. So the table event needs to tell us what to do in the beginning and what to do in the end. Both are optional.

Start: is what defines the Start portion of an event.
Finish: is what defines the Finish portion of an event.

Here is what a very simple html table event might look like.

```
   define event table;
     start:
       put "<table>" NL;
     finish:
       put "</table>" NL;
   end;
```

DIAGNOSTIC TAGSETS, MAKING YOUR LIFE EASIER.
I already mentioned the mapping tagsets for helping you understand tagsets themselves. But there are other tagsets which can help you put the other powers of ODS to use. Namely ODS Styles, and Table Templates. Anything that can make ODS styles easier to deal with has got to be good, right? Hopefully these tagsets will ease your pain.

THE ODS STYLE TAGSET.
All this tagset really does is flatten out the ODS style defined in proc template. In the most basic terms it removes all inheritance and repeats every attribute on every style element that has that attribute defined. The output from this tagset is proc template code that can be run in SAS directly. Here's how you use it.

```
   ods tagsets.ODSStyle
       stylesheet="flat_style.SAS";
```

That's all you need to do. You will get an error message telling you that no body file was specified. But you will get your stylesheet file which in this case is SAS code defining a style called mystyle. I called it mystyle, but you can change the tagset to name the new style whatever you want. You can turn around and run the generated output right off, just do this.

```
   %inc "flat_style.SAS";
```

Here's a short example from that output.

```
proc template;
  define style styles.mystyle;
    style ContentTitle/
      PreText = "Table of Contents" PreHTML =
   '<SPAN onClick="if(msie4==1)expandAll()">'
      PostHTML = '</SPAN><HR size="3">'
      Background = #B0B0B0
      Foreground = #002288
      ContentPosition = left
      Font = ("Arial,Helvetica, Helv", 3, normal
   italic)
      ContentScrollbar = auto
      BodyScrollbar = auto
    ;

    style Table/
      Frame = BOX
      Rules = GROUPS
      Borderwidth = 1px
      CellSpacing = 1
      CellPadding = 7
      Background = #F0F0F0
      Foreground = #002288
      BorderColor = #000000
      ContentPosition = left
      Font = ("Arial,Helvetica, Helv", 3, normal
   normal)
      ContentScrollbar = auto
      BodyScrollbar = auto
    ;

    style HeaderEmphasis/
      Background = #B0B0B0
      Foreground = #0033AA
      ContentPosition = left
      Font = ("Arial, Helvetica, Helv", 3,
   normal italic)
      ContentScrollbar = auto
      BodyScrollbar = auto
    ;

    style NoteContent/
      Background = #E0E0E0
      Foreground = #002288
      ContentPosition = left
      Font = ("Arial,Helvetica, Helv", 3, normal
   normal)
      ContentScrollbar = auto
      BodyScrollbar = auto
    ;
```

For contrast here is what the original ODS style had specified for these style elements.

```
   Style ContentTitle from IndexTitle /
     PreText = text ("content title")
   ;

   Style Table from Output
   ;

   Style HeaderEmphasis from Header  /
     font = fonts("EmphasisFont")
     BackGround = colors ("headerbgemph")
     ForeGround = colors ("headerfgemph")
   ;

   Style NoteContent from Note
   ;
```

As you can see, the flattened style created by our ODSStyle tagset is quite revealing. It sure beats tracing the inheritance through the original proc template code.
Another very nice tagset is style_popup. Style_popup generates html very similar to that created by the htmlcss tagset. The thing that makes it special is that if you are using Internet Explorer to browse the output you can click on the various elements in the output to display the proc template style code which corresponds to that element. As you move your mouse around the screen the item under your mouse will be highlighted in a lovely salmon color. Just run one of your favorite SAS jobs with this ODS statement added in.

```
   ods tagsets.style_popup
       file="body.html"
       stylesheet="style.css";
```

THE STYLE_DISPLAY TAGSET.
Another tagset that is an extension of the style_popup tagset is called Style_Display. Style_Display automatically creates a web page with an example of each type of style element in it, system titles, proc titles, bylines, a table with every kind of cell, notes, everything. It also gives a graphical representation of the inheritance hierarchy used by most of the ODS styles. To create this output you don't even have to run anything just issue the ODS statements.

```
   ods tagsets.style_display
       file="body.html"
       stylesheet="style.css";
   ods tagsets.style_display close;
```

THE NAMEDHTML TAGSET.
Another nice diagnostic tagset is namedHTML. Like these other tagsets namedHTML can trace it's lineage through Style_Popup back to htmlcss. But namedHTML adds object name and label information to the viewable output. If you are familiar with the ODS trace statement or the table templates then you will see the utility of the namedHTML tagset. The output creates a nice viewable output that shows the name and label of each output object.
All of these tagsets that help us with understanding and using ODS are really nice but what we really want to do is make our output look and act the way we want it to. Let's look at some more real world examples of how to do this with tagsets.

CREATING AN MVS PDSE URL.
In this situation we have a problem in the current html output that causes a url on MVS to be created in a way which cannot be interpreted. This is a problem specific to creating html on MVS. The problem is with the quoting. We need to put single quotes around the PDSE. Normal URL's don't have these. It could be argued that IBM needs to fix this behavior, but we can work around it using tagsets.
We can create an entirely new html tagset that fixes that. To keep it simple I created a new event. I called it mvs_src. Mvs_src creates a src attribute with a properly constructed url. I then identified the events that needed fixing up. After that it was a simple matter of replacing the lines that were creating the bad src attribute with a trigger of my new event.

```
proc template;
define tagset tagsets.mvshtml;
 parent=htmlcss;

 /*this is an example of what we want
SRC="HTTP://mvs.SAS.com/MVSDS/'SASCTP.PDSE.HT
ML(CONTENTS)'"
*/

  define event mvs_src;
    put ' src="';
    put BASENAME / if
        !exists(NOBASE);
    put "'" URL "'" /if exists(URL);
    put '"';
  end;

  define event image;
    put "<img";
    putq ' alt=' alt_desc;
    trigger mvs_src;
    putq " border=" BORDERWIDTH;
    put ' usemap="#' @CLIENTMAP;
    put '"' CR / if exists(@CLIENTMAP);
    putq " class=" HTMLCLASS;
    putq " id=" HTMLID;
    put "/>" CR;
  end;

  define event content_frame;
    put '<frame marginwidth="4"';
    put ` marginheight="0"';
    trigger mvs_src;
    put ' name="contents" scrolling=';
    put "auto" / if
        !exists(CONTENTSCROLLBAR);
    put CONTENTSCROLLBAR;
    put ' title="The Table of Contents"';
    put ">" CR;
  end;

  define event body_frame;
    put '<frame marginwidth="9"';
    put ` marginheight="0"';
    trigger mvs_src;
    put ' name="body" scrolling=';
    put "auto" / if !exists(BODYSCROLLBAR);
    put BODYSCROLLBAR;
    put ' title="SAS Output"';
    put ">" CR;
  end;
end;
run;
```

ACTION MAY NOT ALWAYS BRING HAPPINESS; BUT THERE IS NO HAPPINESS WITHOUT ACTION. - Benjamin Disraeli

So what are these trigger things anyway... Triggers allow us to fire off another event. If we are in the start portion of an event then any event we trigger will also do its start. If we are in the finish portion of an event then the triggered event will do it's finish. In the case that the triggered event does not have a start or finish then it will do the statements it has.
We can trigger any event using triggers. Usually the events we trigger are of our own making and do not fall within the set of events that are requested from within SAS. A trigger can also explicitly ask for an event's start or finish. Here is an example tagset that demonstrates triggers.

```
proc template;
  define tagset tagsets.triggers;

    define event doc;
      start:
        put "start of doc" nl;
        trigger mytest;
        trigger jabberwocky;
      finish:
        trigger mytest;
        put "finish of doc" nl;
        trigger mytest start;
        trigger jabberwocky;
        trigger mytest finish;
    end;

    define event mytest;
      start:
        put "start of mytest" nl;
      finish:
        put "finish of mytest" nl;
    end;

    define event jabberwocky;
      put "This is Jabberwocky" nl;
    end;

  end;
run;
```

Notice that we use the doc event. The doc event is the first event that ODS requests for the body file. This event is requested as soon as the file is opened, which is the same time as the ODS MARKUP statement. Creating output from this tagset is as easy as it was for the style display tagset.

```
ods tagsets.triggers
    file="body.txt";
```

```
ods _all_ close;
```

The output created from this would be:

```
start of doc
start of mytest
This is Jabberwocky
finish of mytest
finish of doc
start of mytest
This is Jabberwocky
finish of mytest
```

USING AN EXISTING CASCADING STYLESHEET DEFINITION.

Here's a case where we didn't want to use ODS styles at all. We already had a set of cascading stylesheets that were used for everything else. Many of the stylesheet entries were based on tag names like <h2>, <h3>, etc. But a few of them had names that needed to be specified as the class attribute on the tag. In this case all of those tags were the ones used to create tables. There are ways to do that using the style template. But you can also do it using tagsets. This is a basic example for a stylesheet, which only has entries for the components of a table. It was also desirable in this instance to fix the attributes of the table so that they would not be affected by ODS styles. Notice that we use the phtml tagset as our parent in this case. Phtml is different from htmlcss in that it uses a limited stylesheet that uses tag names exclusively. Phtml also uses h1 for its titles. Those had to be changed to h2 so they would pick up the right definition from the stylesheet. There are no class attributes generated by the phtml tagset that makes it ideal for this situation.

```
proc template;
define tagset tagsets.mycss;
  parent=tagsets.phtml;

  define event system_title;
    put "<h2";
    trigger align;
    put ">";
    put VALUE;
    put "" CR;
  end;

  define event system_footer;
    put "<h2";
    trigger align;
    put ">";
    put VALUE;
    put "" CR;
  end;

  define event table;
    start:
      put "<div";
      trigger align;
      put ">" CR;
      put "<table";
      putq " border=" BORDERWIDTH;
      put ' border="1"' / if
          !exists(BORDERWIDTH);
      putq ' cellspacing= "0"';
      putq ' cellpadding="5"' ;
      putq " rules=" RULES;
      putq " frame=" FRAME;
      put ' class="main_table"';
      put ">" CR;
    finish:
      put "</table>" CR;
      put "</div>" CR;
      put "" CR;
  end;

  define event header;
    start:
      put "<th";
      trigger rowcol;
      trigger headalign;
      put ' class="thcell"';
      put ">";
      put VALUE;
    finish:
      put "</th>" CR;
  end;

  define event data;
    start:
    /* this is so we'll get a header when we
       are supposed to. Even if the proc
       isn't telling us it's a header. */
      trigger header /if cmp(section,"head");
      break /if cmp(section, "head");
      trigger header /if
              cmp(htmlclass, "rowheader");
      break /if cmp (htmlclass,"rowheader");
      put "<td";
      trigger rowcol;
      trigger align;
      put ' class="tdcell"';
      put ">";
      put '<pre>' /if exists(asis);
      put VALUE;
    finish:
      trigger header /if cmp(section,"head");
      break /if cmp(section, "head");
      trigger header /if
              cmp(htmlclass,"rowheader");
      break /if cmp(htmlclass, "rowheader");
      put '</pre>' /if exists(asis);
      put "</td>" CR;
  end;
end;
run;
```

To run this we issue an ODS statement something like this.

```
ods tagsets.mycss file="body.html"
stylesheet=(url="http://myserver/standard.css
");
```

This will not create a stylesheet file but it will create a link to your existing stylesheet.
After that example I'm betting you have a lot of questions. We've already seen triggers, and puts but what about those if's?

ARGUE FOR YOUR LIMITATIONS, AND SURE ENOUGH, THEY'RE YOURS. - Richard Bach

CONDITIONALS, IF AND WHEN.

Conditions can be added to any event statement. They are always preceded by a slash. And they only affect the line they are on. Exists and any can have one or more arguments. Cmp takes 2 or more.

IF AND WHEN, OR NOT.

If and when are interchangeable and optional. The important part is the slash. All of these are equivalent. If's can compare values and strings, or check variables for value.

```
   put "foreground has a value!!!" NL / if
       exists(foreground);
   put "foreground has a value!!!" NL / when
       exists(foreground);
   put "foreground has a value!!!" NL /
       exists(foreground);
```

COMPARE
Cmp compares a list of variables and strings for equality.

```
put "The foreground is blue!!" nl / if
    cmp("blue", foreground);
```

ANY OF THEM
Any checks a list of variables for values. If any of them have a value then it is true.

```
put "Some variables have a value!!" nl / if
     any(background, foreground, cellpadding,
     cellspacing);
```

ALL OF THEM
Exists checks a list of variables for values. If all of them have a value then it is true.

```
put "All variables have a value!!" nl / if
     exists(background, foreground, cellpadding,
     cellspacing);
```

All conditions can be negated with a '!' or '^'

```
   put "The foreground is not red!!" nl /
       if !cmp("red", foreground);
```

BREAKS.
You can cause an event to stop execution by inserting a break. Any statements below the break will not be executed. This is most useful when combined with conditionals. The following example will probably make your head hurt. This is the event code we use to create the horizontal and vertical justification values for everything in our html destinations. Horizontal alignment defaults to left or 'l' and vertical alignment defaults to center or 'c'. So we don't want to print those because they are unnecessary. The output which these events will create are things like id="r", id="rb", id="c". I'll let you figure out the rest.

```
define event align;
  break / when !any(JUST, VJUST);
  trigger valign / when !exists(JUST);
  trigger halign / when !exists(VJUST);
  break / when !exists(JUST, VJUST);
  trigger valign / when cmp("l", JUST);
  trigger halign / when cmp("c", VJUST);
  break / when cmp("l", JUST);
  break / when cmp("c", VJUST);
  put ' id="' JUST;
  put VJUST;
  put '"';
end;

define event halign;
  putq ' id=' JUST / when !cmp("l", JUST);
end;

define event valign;
  putq ' id=' VJUST / when !cmp("c", VJUST);
end;
```

SO YOU REALLY WANT GREEN BARS?
Unlike the align events above, these events were written after I created user variables. The resulting code is a lot easier to understand. Have you ever wanted to create a table with alternating row colors? Well now you can. We'll create two new style elements and then switch between them on every row. We need to redefine the header and data events. You could either remove class= from them or have their class= point to your alternating class name. If you put class on the <tr> you don't need it on the cells. A less elegant way would be to just swap the color instead of the class name, but then you'll have to change the tagset every time you want to change the way it looks.

```
   proc template;
   define Style styles.gbdefault;
   parent = styles.default;

     style dark from Data/
        background = cx00dd00
     ;
     style light from Data
     ;
   end;
   run;

   proc template;

   define tagset tagsets.htmlalt;
     parent=tagsets.htmlcss;

     define event shortstyles;
       trigger bodystyle;
       trigger titlestyle;
       trigger proctitlestyle;
       trigger tablestyle;
       trigger tdstyle;
       trigger thstyle;
       trigger tdstyle2;
       trigger tdstyle3;
     end;

     define event tdstyle2;
       style="light";
       put "light {" CR;
       trigger stylesheetclass;
       put "}" CR;
     end;

     define event tdstyle3;
       style="dark";
       put "dark {" CR;
       trigger stylesheetclass;
       put "}" CR;
     end;

     define event table ;
       start:
         set $rowclass "dark";
         put '<table>' nl;
       finish:
         put '</table>' nl;
     end;

     define event row;
       start:
         trigger swapclass;
         put '<tr';
       putq ' class=' $rowclass;
       put '>';
       put nl;
     finish:
       put '</tr>' nl;
     end;
```

```
      define event data;
        start:
          put '<td';
          putq ' class=' $rowclass;
          put '>';
          put value;
        finish:
          put '</td>' nl;
      end;

      define event swapclass;
        unset $swapped;
        set $swapped "1" /if
            cmp($rowclass,"dark");
        set $rowclass "light" /if
            cmp($rowclass, "dark");
        break /if exists($swapped);
        set $rowclass "dark" /if
            cmp($rowclass, "light");
      end;
    end;

    run;

    ods tagsets.htmlalt style=gbdefault
        file="greenbar.html"
        stylesheet="greenbar.css";
    proc print data=sashelp.class;run;
    ods _all_ close;
```

That wasn't too bad was it. Ninety percent of the things people are doing with tagsets fall into the same category as these examples. These are the easiest kind of changes to make. But some of you will want more. The pioneering spirit has brought us several new tagsets, two xhtml tagsets, SYLK, a new LaTeX tagset, even one that talks directly to excel using DDE!

TAGSET ATTRIBUTES.
There is more you should know before you go forging your own path. We've already seen default_event. But there are others. Markup languages generally have a set of control characters which it uses to define it's commands. In HTML at the very least those are <>. In Latex it is primarily the \. In order for the tagset to treat these characters properly it needs to know about them. Tagsets have two attributes that control this. map and mapsub.

THE CHARACTER MAP
The character map is simply a string of characters to look out for. A simple string for HTML might be '<>&'

THE MAP SUBSTITUTION STRINGS
The substitution strings correspond to the character map. A special character of your choosing separates the strings. In this case I'm going to use '/'. So, the strings corresponding to the character map above
would be '/&lt;/&gt;/&amp;/'

Some markup languages have non-breaking spaces. It's white space that cannot be broken between lines. You can use the nobreakspace attribute to specify the string which defines a non-breaking space.

If your target markup language has such a thing as a non-breaking space you can define it. ODS will then make good use of it when it can. For the HTML tagset you would use

```
nobreakspace = '&nbsp;';
```

SPLITTING LINES
You may also tell ODS what it should use to break lines. For our HTML example we would do this: split = '<br>'

EMBEDDED STYLESHEETS
I didn't originally intend for tagsets to allow for in lined stylesheets. It just seems wrong to me. But a lot of people wanted it so I added an option. Setting embedded_stylesheet to yes will cause ODS to request the stylesheet events in the head section of each file. But this behavior only occurs if the stylesheet argument was missing from the ODS statement.

OUTPUT TYPE.
Each tagset has an output type. The default for this is XML. But you don't want SAS thinking your HTML or CSV output is XML. SAS won't know how to display your output. In the case of LaTeX the tagsets actually behave differently. Among other things colors are formatted differently.

ALTOGETHER.
So all together a basic tagset definition might look something like this.

```
define tagset tagsets.mytags;
map = '<>&';
mapsub = '/&lt;/&gt;/&amp;/';
nobreakspace = '&nbsp;';
split = '<br>';
embedded_stylesheet = yes;
output_type = html;
```

EVENT ATTRIBUTES.
Events can be controlled to some extent, there only purpose isn't just to contain put statements. Events can be directed to write to any of the files specified on the ODS statement. They can also specify a style element to use.
You can redirect any event to any of the known output files that are open. If the user didn't specify that file on their ODS statement then it will behave as if the event was not defined. The names of those files correspond to names on the ODS statement itself. The Logical names are , body, frame, contents, pages, code, data, and stylesheet. All files remain open until the ODS close statement is encountered. So, to redirect our event's output to the content file we would add this line to your event.

```
file=contents;
```

A style can also be specified for an event. The style must be defined in the ODS style definition that is currently in use.

```
style="mystyle";
```

It is also sometimes desirable to over-ride the stylesheet behavior that prevents some style attributes from being surfaced in the tagset. To get all the attributes of a style element regardless of them being defined in the stylesheet use the pure_style option.

```
Pure_style=yes;
```

Altogether, To create an event that over-rides stylesheet behavior, prints to the contents file and uses the mystyle element from the ODS style. Do something like this.

```
define event myFullStyleEvent;
    file=contents;
    style="mystyle";
    pure_style=yes;
```

LOOKS AREN'T EVERYTHING BUT THEY CERTAINLY HELP
Tagets will also do indention for you. This can be quite handy when looking for errors in your tagset. You can bet that something is wrong with your tagset when the indention in your output goes wrong. You can specify the indention depth for a tagset.

```
    Indent=2;
```

Now, every time an ndent or xdent statement is encountered the output will indent or out dent 2 spaces.

INDENTING.
Ndent indents the output one more indention level

OUT DENTING.
Xdent out dents the output one less indention level.

Adding indention to our previous trigger example makes it much easier to understand.

```
define tagset tagsets.test;
indent = 2;

define event doc;
start:
put "start of doc" nl;
ndent;
trigger mytest;
trigger jabberwocky;
finish:
trigger mytest;
xdent;
put "finish of doc" nl;
trigger mytest start;
trigger jabberwocky;
trigger mytest finish;
end;

define event mytest;
start:
put "start of mytest" nl;
ndent;
finish:
xdent;
put "finish of mytest" nl;
end;

define event jabberwocky;
put "This is Jabberwocky" nl;
end;

The output created from this would be:

start of doc
start of mytest
This is Jabberwocky
finish of mytest
finish of doc
start of mytest
This is Jabberwocky
finish of mytest
```

The very essence of the creative is its novelty, and hence we have no standard by which to judge it. Carl R. Rogers

CREATING YOUR OWN TAGSET.
If you wish to create your own tagset completely from scratch a good way to start is to inherit from a tagset that is similar to what you hope to create. If there is nothing similar then the event_map or text_map tagset may be what you need. As you create new events they will take the place of the default. When you get it the way you want it then remove the event_map inheritance. Your output will only reflect the events you defined.
If you choose a parent tagset that doesn't have a default event then start with a tagset definition like this one. If your parent does define a default event then simply leave the default event out of your tagset. The real purpose of having a default event in this case is so you can see the events that you haven't defined yet. You may want to use some of them. But how can you if you don't know they are there?

```
proc template;
define tagset tagsets.mynewtagset;
parent=tagsets.phtml;
default_event="mydefault";

define event mydefault;
start:
put '<' event_name '>' nl;
finish:
put '</' event_name '>' nl;
end;
end;
run;
```

Run a simple SAS job using your new tagset.

```
ods tagsets.mynewtagset file="body.html";
proc print data=SAShelp.class;run;
ods _all_ close;
```

The first time you look at the contents of body.html you will see things you never want to see again. For every event you don't want to see create an empty event at the bottom of your new tagset that looks like this one. We can remove these later so it's a good idea to keep them together. In this example we are getting rid of the seldom used cellspecsep event.

```
define event cellspecsep;
end;
```

Identify the events that you do want. Create new definitions to put them into the shape you want them. Eventually you won't see any text in your output that was generated by the default event. When that happens you can do a few different things.

1. You can remove the parent specification on your tagset. But only if you aren't using any of it's events.
2. You can remove your default event and all of the empty events you created to block the default event.

CONCLUSION
At this point you should have an idea of how easy it really can be to change the markup output generated by ODS and the libname engine. You may already be thinking about what you want to change. Through the abilities of ODS MARKUP and Tagsets I hope that I have given you the power to keep up with the unreasonably fast paced world of markup language output.

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

Eric Gebhart
SAS Institute
SAS Campus Drive
Cary NC 27513
Work Phone: 919-531-5817
Email: Eric.Gebhart@SAS.com
Web: http://ww.SAS.com