Paper 039-2008

A Tiptoe Through the Tagset Field

Michael Molter, PPD Inc.

ABSTRACT

The flexibility that the markup destination has added to ODS is comparable to the benefits ODS originally brought to SAS® output, but it comes at a price known as the tagset. With ODS, we could display results in web pages without HTML knowledge; in Word without RTF knowledge. Developers soon realized though that having ODS do all the work wasn't always ideal; that on occasion, users could benefit from having some control over how markup was generated. Just as templates instruct ODS on display customization, tagsets instruct ODS on markup customization. The advantages are clear – even more flexibility in the generation of output. The price is similar to that paid for display flexibility – more TEMPLATE procedure syntax, and an understanding of how ODS constructs files from these instructions.

Great accomplishments often come from modest beginnings. This paper is your modest beginning. From the edge of the field we'll get "the lay of the land" with discussions on background, purpose, and use with the Markup destination. With one foot in the field we'll look at examples of markup that require slight tweaking and available documentation to guide us. Further into the field, we'll produce simple informational tagsets, and inch out further with tagsets that inherit and manipulate those that SAS provides. By the end of the paper, the reader will see in the distance many undiscovered areas, but hopefully the curiosity generated by the path taken thus far will lead to further exploration and bridges to greater accomplishments.

INTRODUCTION

Let's face it, the Output Delivery System (ODS) has spoiled us, and in many ways. For starters, there was flexibility in file formats. For years, the use of long DATA steps with lots of PUT statements was how output was delivered to word processing or spreadsheet programs. When the output came from PROCs, this took place only after the output was collected from the PROC, manipulated, and organized in a data set in a way that optimized the delivery of the output. Significant trial and error in order to place the output in the exact right spot was also common. An alternative was to use the ASCII-based output file that SAS produced. With ODS and these "things" called "destinations", we could, with some simple syntax, have the best of both worlds – we could use the PROC output produced by SAS, but deliver it, instead of to an ASCII file, to other formats such as Word, PDF, HTML, and others. In addition, ODS gave us something we never had before – control over certain aspects of the output – particularly, how a table was laid out, and the appearance or style attributes of the file. With some knowledge of the TEMPLATE procedure and how a file or a table was broken into "pieces" (elements), we could create global templates that, when specified as part of the simple ODS syntax, would define how the file looked. Additionally, some PROCs allowed for "inline" styling that would remove the need for a global template. With the ability to deliver SAS output to different file formats without tedious DATA steps, and the additional freedom to customize its appearance, the world of SAS output was a better place to be.

CONTROL VS. CONVENIENCE

The beauty of it all was in what I call the "magic of ODS." All we had to do was to learn more SAS syntax. We could create and customize an HTML file without knowing anything about HTML; a Word file without knowing about RTF. By creating one global template with syntax defined by SAS, ODS knew how to translate it into the language of our choice. It's the same convenience we enjoy when we go to a restaurant and pay someone to cook our dinner and bring it to us, or when we pay someone to fix our car when it needs repairs. Not only does it save us work, but sometimes we may be letting someone more knowledgable than ourselves complete the necessary task.

One person's convenience, though, is another person's loss of control. While one is comforted by the fact that an expert mechanic is fixing their car, another feels helpless handing the reigns over to someone else - not knowing exactly how each decision is made, what the quality and the cost of each part is, how much caution is being exercised. By creating and then using style and table templates or inline styling, we are asking ODS to translate PROC TEMPLATE (or other procedures) syntax into the destination language in the area of the file where styles or tables are created. Other areas unrelated to styles and tables are untouchable. To many, allowing ODS to be the "mechanic" is all they need, but another group of programmers has evolved since the originial days of ODS. These programmers prefer to bypass the "interpreter", "speak" in the native language themselves, and have access to any part of the file. They sacrifice the convenience of a tool that works for all destinations in favor of infinite control of a destination of their choice. The tagset is their means for gaining this control.

SCOPE

This paper is intended to get your feet wet by introducing the tagset to you as a means of fixing the flaws in your ODS output. The object of this paper is a soft, gentle introduction to tagsets. Most SAS users have a need for procedure output, and yet tagsets and the Markup destination are still in relatively early stages of development. Many users have heard nothing about tagsets. Many who have, know of a reputation of being difficult to program. Putting these

facts together, it's reasonable to wonder how many programmers are producing less-than-ideal output that they don't realize can be customized or don't know how to. In this paper we'll discuss the basic concept and structure of a tagset. Moving past the abstract and into the concrete, we'll then look at examples of output from tagsets supplied to us from SAS, identify candidates for customization within each, and develop resources (including other tagsets) to help us identify what part of the tagset was responsible for those areas. With some background on terminology and syntax, we'll inherit the original tagset and modify it to produce the desired text.

Tagsets offer us something we've never had before – control over any or all of the text (markup) that makes up the file being produced. As you can imagine, this has broad implications, well beyond this paper's scope of customizing ODS output. For the most part, we will stick to discussions about statements, options, syntax, variables, and other tools that are relevant to our examples. We will also be focusing mostly on making small modifications to pre-existing tagsets through inheritance. Tagsets that are built "from scratch" or almost from scratch, such as those supplied by SAS, require a thorough knowledge of the steps that ODS takes to create a file and the relationship between ODS and the PROC. Though this will be briefly discussed in this paper, a detailed discussion is beyond our scope. Hopefully, such knowledge can be accumulated with experience. Finally, even further beyond our scope, as we move into SAS version 9.1 and later 9.2, tagsets begin to form the basis of the new graphic templates. Beyond ODS, they are used with the XML Libname engine as well as the creation of files from the CDISC procedure.

You will also notice that most of the examples use HTML. This is not to suggest that HTML is the only markup tagsets can generate. On the contrary, text that constitutes valid RTF that Word can read, or valid XML that Excel can read, or any other markup, can be generated just as easily in the same way that HTML is generated. Version 9.1 however does not have usable RTF tagsets from which we can inherit. While a discussion of HTML or any other markup is beyond our scope, the reader that is unfamiliar with HTML will be glad to know that the examples in this paper require very little if any HTML knowledge. What is necessary to know will be explained. The examples in this paper assume version 9.1.3.

CONCEPTS

What do you think of when you think of the PRINT procedure? What kind of display do you associate with code such as the following?

```
proc print noobs data=sashelp.class(obs=10) ;
var name sex age height weight ;
run;
```

Most likely, you picture something that resembles Figure 1.

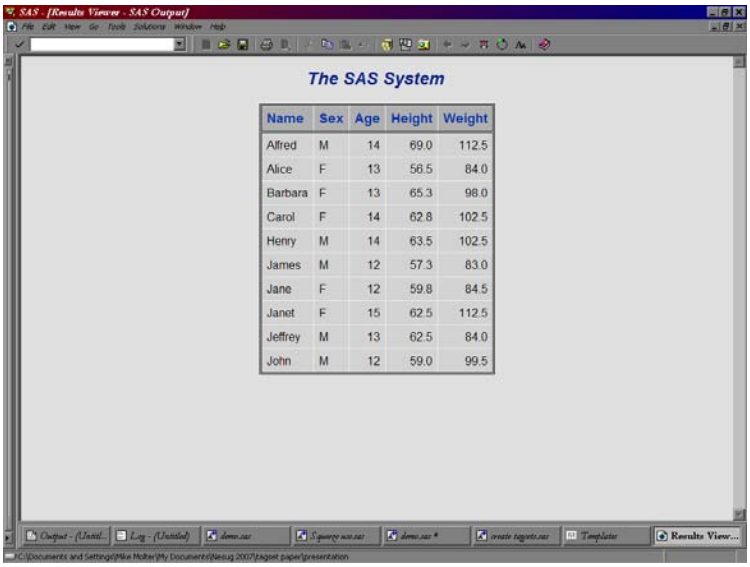
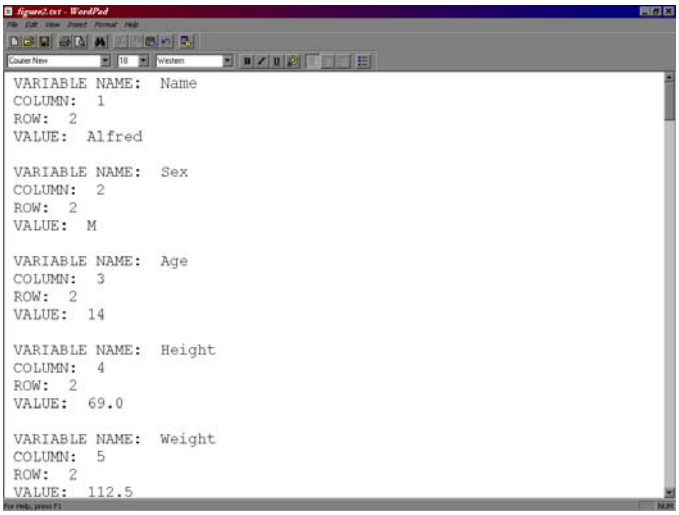


Figure 1

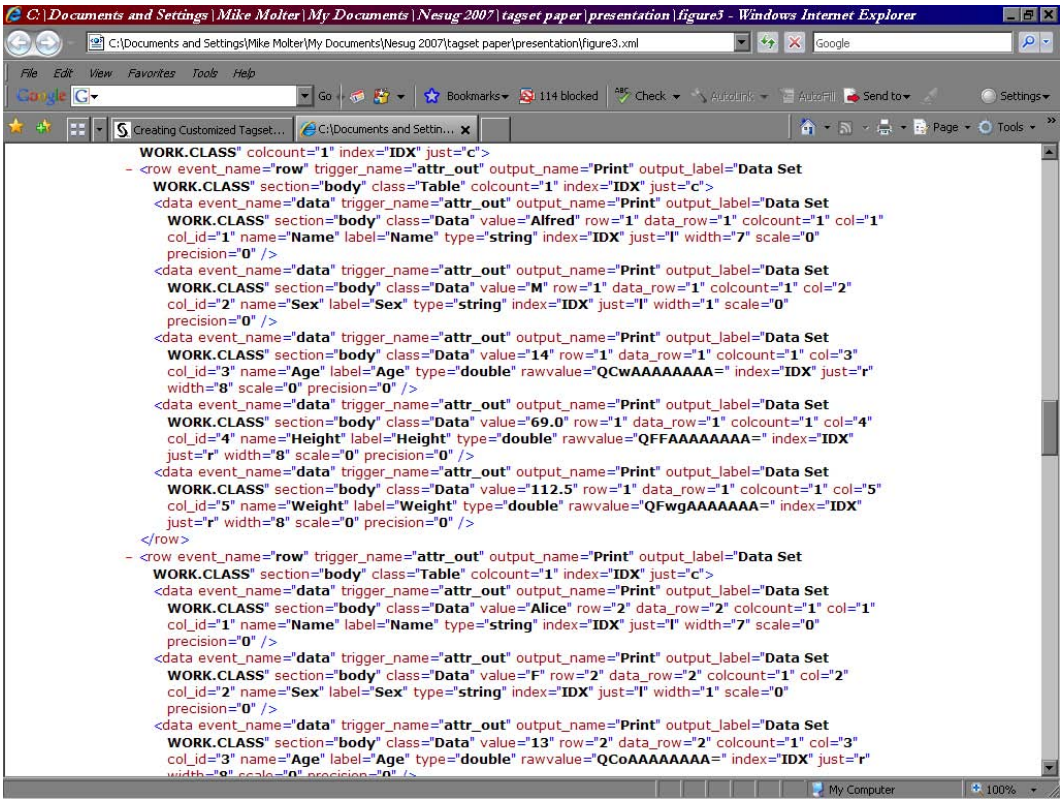
And why wouldn't you? Everything you can do in PROC PRINT is designed with this type of output in mind. The VAR statement allows you, the user, to decide what the columns are, or, put another way, how data is laid out horizontally across a page. Style attribute syntax (not seen in this example) controls aspects of the report such as colors, fonts, column width, and cell alignment.

Counter to every pre-conceived notion about what PROC PRINT output should look like, Figures 2 through 4 also illustrate snippets of output from the code above. How can this be? Let's first look at Figures 2 and 3.



```
VARIABLE NAME:  Name
COLUMN:  1
ROW:  2
VALUE:  Alfred

VARIABLE NAME:  Sex
COLUMN:  2
ROW:  2
VALUE:  M

VARIABLE NAME:  Age
COLUMN:  3
ROW:  2
VALUE:  14

VARIABLE NAME:  Height
COLUMN:  4
ROW:  2
VALUE:  69.0

VARIABLE NAME:  Weight
COLUMN:  5
ROW:  2
VALUE:  112.5
```

Figure 2



```
WORK.CLASS" colcount="1" index="IDX" just="c">
- <row event_name="row" trigger_name="attr_out" output_name="Print" output_label="Data Set
  WORK.CLASS" section="body" class="Table" colcount="1" index="IDX" just="c">
  <data event_name="data" trigger_name="attr_out" output_name="Print" output_label="Data Set
    WORK.CLASS" section="body" class="Data" value="Alfred" row="1" data_row="1" colcount="1" col="1"
    col_id="1" name="Name" label="Name" type="string" index="IDX" just="l" width="7" scale="0"
    precision="0" />
  <data event_name="data" trigger_name="attr_out" output_name="Print" output_label="Data Set
    WORK.CLASS" section="body" class="Data" value="M" row="1" data_row="1" colcount="1" col="2"
    col_id="2" name="Sex" label="Sex" type="string" index="IDX" just="l" width="1" scale="0"
    precision="0" />
  <data event_name="data" trigger_name="attr_out" output_name="Print" output_label="Data Set
    WORK.CLASS" section="body" class="Data" value="14" row="1" data_row="1" colcount="1" col="3"
    col_id="3" name="Age" label="Age" type="double" rawvalue="QCwAAAAAAAA=" index="IDX" just="r"
    width="8" scale="0" precision="0" />
  <data event_name="data" trigger_name="attr_out" output_name="Print" output_label="Data Set
    WORK.CLASS" section="body" class="Data" value="69.0" row="1" data_row="1" colcount="1" col="4"
    col_id="4" name="Height" label="Height" type="double" rawvalue="QFFAAAAAAAA=" index="IDX"
    just="r" width="8" scale="0" precision="0" />
  <data event_name="data" trigger_name="attr_out" output_name="Print" output_label="Data Set
    WORK.CLASS" section="body" class="Data" value="112.5" row="1" data_row="1" colcount="1" col="5"
    col_id="5" name="Weight" label="Weight" type="double" rawvalue="QFwgAAAAAAA=" index="IDX"
    just="r" width="8" scale="0" precision="0" />
  </row>
- <row event_name="row" trigger_name="attr_out" output_name="Print" output_label="Data Set
  WORK.CLASS" section="body" class="Table" colcount="1" index="IDX" just="c">
  <data event_name="data" trigger_name="attr_out" output_name="Print" output_label="Data Set
    WORK.CLASS" section="body" class="Data" value="Alice" row="2" data_row="2" colcount="1" col="1"
    col_id="1" name="Name" label="Name" type="string" index="IDX" just="l" width="7" scale="0"
    precision="0" />
  <data event_name="data" trigger_name="attr_out" output_name="Print" output_label="Data Set
    WORK.CLASS" section="body" class="Data" value="F" row="2" data_row="2" colcount="1" col="2"
    col_id="2" name="Sex" label="Sex" type="string" index="IDX" just="l" width="1" scale="0"
    precision="0" />
  <data event_name="data" trigger_name="attr_out" output_name="Print" output_label="Data Set
    WORK.CLASS" section="body" class="Data" value="13" row="2" data_row="2" colcount="1" col="3"
    col_id="3" name="Age" label="Age" type="double" rawvalue="QCoAAAAAAAA=" index="IDX" just="r"
```

Figure 3

By comparing each of these to Figure 1 above, you'll notice (if you look carefully) that the data is there. The problem is that despite telling PROC PRINT exactly how we want columns laid out with the VAR statement, neither of these outputs has columns, or at least, columns that are in one-to-one correspondence with the variables listed in the VAR statement. Figure 4 is even more counter-intuitive.

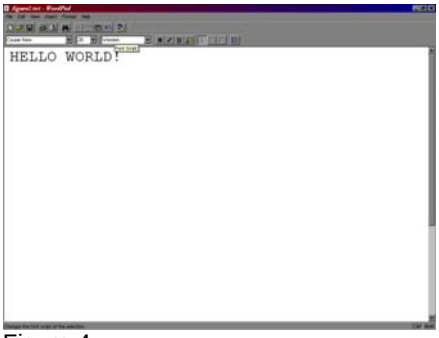


Figure 4

Not only are there no columns, there isn't even any data! The statements whose purpose is to lay out the display of the table appear to be having no effect.

Our surprise at the results illustrated by Figures 2 thru 4 speaks to our natural tendency to confuse the roles of ODS with the PROC. These figures help point us in the right direction. The job of the PROC is to analyze the data, and in the case of PROC PRINT, to lay out the table. Like an advisor to the president, the PROC then delivers this information to ODS. Of course the president has several advisors. Continuing with this analogy, ODS has input coming from other sources such as global statements (e.g. TITLE statements, OPTIONS statements, ODS statements, etc.). In the end, just like the president, the decision rests with ODS as to how to use this information, and ultimately, as its name implies, to deliver the output. In the early days of ODS, this was done with the magic of ODS – a gap in our knowledge about the spectrum that starts with raw data and ends with deliverable output. Now, and more so in the future, the tagset is available for users to provide instructions to ODS on how to use and deliver the output.

Incidentally, let's make sure we understand exactly what this "output" is. Let's return to Figure 1. We're tempted to say that this is produced by ODS, but the truth is that this is produced by a program such as Internet Explorer, Microsoft Word, or Adobe Acrobat that reads the "real" file and displays it in its own way. To see this "real" file, open the same file with Notepad. The text you see in this view of the file is called ***markup***. Figure 5 illustrates a snippet of the HTML markup that Intenet Explorer interprets as Figure 1. Markup is a text file that contains not only the text you see through the viewing program (e.g Internet Explorer), but also special instructions called ***tags*** that don't show up in the viewing program's view, but that the viewing program uses to display the file. HTML uses angle brackets to surround its tags. A ***tagset*** is a SAS file that provides instructions to ODS on how to generate text. To customize something like HTML, we simply customize the markup being generated.

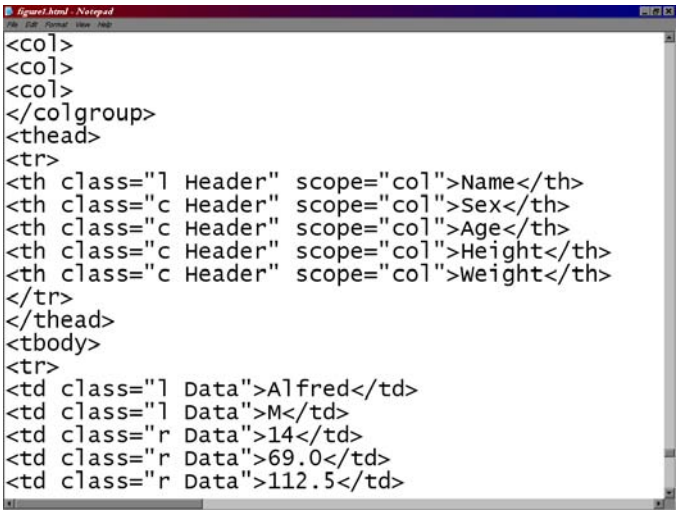


Figure 5

THE EVENT MODEL

So delivering output really means generating text, but what do we mean by "instructions to ODS on how to use and deliver output"? If all we're dealing with is a text file, then all we need is FILE and PUT statements, right? With this approach, however, we end up building the file manually by ourselves – we provide all the markup to create a table, its column headers, the correct number of rows and columns, even the cell content itself. All of this is information generated by the PROC, but with this approach, we can only capture it by running the PROC, dumping the information to a data set, and then reading it with a DATA step. Tagsets, on the other hand, allow us to define general rules for generating separate pieces of the text. For example, if we're creating a table that has five columns,

we define in the tagset a general rule for creating the markup that defines a column, and ODS knows (from the PROC) to apply that rule five times, and when to apply it. Each of these rules is called an ***event***. ODS creates a file in a sequence of discrete steps, each of which calls an event. When the ODS statement is issued, a certain boiler plate of events is called, starting with INITIALIZE, then DOC, and a few more. After that, the PROC is processed and more events are called. These events transfer information such as data values, column and row numbers, style attributes, etc, from the PROC to the markup by way of ***event variables***. Just as PUT statements in a DATA step write text as a combination of literal text and the values of data set variables, the tagset event uses PUT to write markup as a combination of literal text (e.g. tags) and information passed to the event by the PROC through event variables.

The tagset is a mechanism for the modular definition of output generation that allows us to take full advantage of information gathered by the PROC while it is processing. "Modular definition" here refers to the event model – defining each piece of output independent of each other. A tagset is simply a collection of events that together define the entire file. Its modular nature allows us to modify one piece without affecting the others. This is done through inheritance and redefining an event in the same way we inherit style templates to modify individual style elements. "Taking full advantage of information gathered by the PROC while it is processing" means that through the event, we simply tell ODS "here's how to generate this text." When to generate it and how often is left up to ODS and the PROC. Additionally, the text itself doesn't all have to be literal. By supplying event variable names in the PUT statements in the same way we supply data set variable names in PUT statements in a DATA step, we can generate dynamic text that only the PROC knows. Maybe most importantly, this can be done while the PROC processes, rather than with a DATA step after the PROC has finished processing.

CHALLENGES AHEAD

We've discussed the event in theoretical terms, but to use the event to tell ODS what to do at certain steps in the process requires something more concrete. What are the names of the events, and in what order are they used? Armed with this knowledge and central to our efforts in this paper, how do we associate a part of the output with an event? Once we have that, what tools are at our disposal? What are the names of the event variables? For which events does any given variable carry useful information, and for how long is this information available? In the following sections we'll use the answers to these questions to begin building our own tagsets.

BUILDING THE TOOLBOX

As is the case with table and style templates, tagsets are stored in template stores. A template store is a unit of storage specifically for templates and tagsets that, like Windows directories, allows for directory structures within itself. SAS ships two template stores for our use: TMPLMST, a read-only store in the SASHELP library that contains all the SAS-supplied tagsets in a directory called TAGSETS; and TEMPLAT, a read-write store in the SASUSER library. We use the ODS PATH statement to tell ODS which store to either look for a tagset we are trying to use or save a tagset we are trying to create (unless overridden by the STORE= option on the DEFINE TAGSET statement). Each store is specified as a two-level name beginning with the *libref* associated with the directory in which the store is located, followed by a period and then the name of the store. If multiple stores are listed, ODS will search for tagsets in stores in the order in which they are listed. When creating a tagset, if the first write-access store listed on the ODS PATH statement doesn't exist, the store will be created. When no ODS PATH statement is used, the implied order is SASHELP.TMPLMST followed by SASUSER.TEMPLAT. One can view a tagset definition (SAS-supplied or user-created) by clicking in the Results window in PC SAS, clicking the View menu, and choosing Templates. The template stores will be listed in the left frame. By expanding the SASHELP.TMPLMST store, one will see folders, most of which contain table templates, but one, Tagsets, that contains tagset definitions. By expanding this folder and double-clicking any of the tagsets on the right side, the definition can be browsed.

USING TAGSETS

Three different methods exist for using a tagset for ODS output, one of which you may have used before without knowing it. Certain ODS destinations call a tagset behind the scenes. Examples of such destinations include PHTML, HTML4, and CSV.

A second method is available when your tagsets are stored in subdirectories within the template store. When this is the case, you can specify the path within the store as if it were another destination. Consider the following statements.

```
libname mypath "c:\my documents" ;
ods path sashelp.tmplmst(read) mypath.tagstore(update) ;
ods mytags.tag1 file="using_tag1.html" ;
```

In the third statement, SAS is looking for a tagset called TAG1 in a folder called MYTAGS. It first looks for it in the TMPLMST template store located in the SASHELP directory. Not finding it there, it then looks in the template store TAGSTORE located in the MYPATH directory in My Documents.

The third method uses MARKUP as the name of the destination, specifying the tagset in the TAGSET= option, as in the following.

```
ods markup tagset=mytags.tag1 file="using_tag1.html" ;
```

Slight differences exist between the second and third methods. The second method requires the tagset to be stored inside a folder (or any level of subfolders) within the template store. ODS MARKUP does not have this requirement. If TAG1 was not contained within any folder of the TAGSTORE template store, we could specify TAGSET=TAG1 on the ODS MARKUP statement. On the other hand, without the ID= option set, ODS only allows us to have one instance of any particular destination open at a time. Suppose we want to send output to multiple files with different tagsets. Consider the following.

```
ods phtml file="file1.html" ;
ods mytags.tag1 file="file2.html" ;
ods mytags.tag2 file="file3.html" ;
proc print data = sashelp.class ; run ;
ods _all_ close ;
```

The above code is valid because we have three "destinations" open. Suppose, however, that TAG1 and TAG2 are not contained in directories within the template store, forcing us to use the MARKUP destination to produce FILE2 and FILE3. The MARKUP destination can only be open for multiple instances if the ID= option is specified as below.

```
ods phtml file="file1.html" ;
ods markup (id=f2) tagset=tag1 file="file2.html" ;
ods markup (id=f3) tagset=tag2 file="file3.html" ;
proc print data = sashelp.class ; run ;
ods _all_ close ;
```

WRITING TAGSETS

As mentioned earlier in the scope, the tools we discuss will, for the most part consist only of those needed to modify markup in the examples. At this point we'll discuss those tools in general terms. Each tool will then be discussed in more detail when the time comes to use it. We begin by looking at the structure of a tagset.

As with other aspects of the tagset, its structure is similar to that of a style template definition within PROC TEMPLATE, as seen below.

```
PROC TEMPLATE ;
DEFINE TAGSET tagset-name ;

tagset statements

DEFINE EVENT 1;
event statements
END ;

DEFINE EVENT 2;
event statements
END ;

etc

END ;
RUN ;
```

The first statement after the PROC statement is the DEFINE TAGSET statement. *Tagset-name* can be a one-level name or a multi-level name in which levels are separated by periods and represent directories in the store. For example, in the statement *define tagset level1.level2.tag1*, *level1* represents a top-level directory in the template store, *level2* represents a sub-directory of *level1,* and tag1 is the name of the tagset stored in the *level2* subdirectory. If the directories don't exist in the store, they will be created. As mentioned earlier, the store to which the tagset is saved is the first one on the ODS PATH statement that has write-access. This can be overridden by specifying */ store*= followed by the name of a store (an existing one or one to be created) immediately after the tagset name.

Following the DEFINE TAGSET statement and before the event definitions are the tagset statements or tagset *attribute* statements. Three of the more common statements are the INDENT=, PARENT= and the DEFAULT_EVENT= statements. INDENT= specifies how many spaces to the right text is to be moved when the NDENT statement is found within an event definition, and spaces to the left when an XDENT statement is found. PARENT= specifies the name of a tagset (found by searching the stores in the ODS PATH statement) to inherit.

Events in the "child" tagset override events of the same name in the parent tagset. When an event is called that is not defined in the child tagset, the parent tagset is checked. When an event is requested that doesn't exist anywhere, nothing happens. On the other hand, we can use DEFAULT_EVENT= to specify an event that will be used in the absence of the event that was called. We'll see soon that this can be a useful tool in learning about which events are used by which PROCs and in which order they are used.

We now move into the event definition and begin with an event's state. Because of the hierarchical nature of markup, events that produce markup are often defined with a ***start state*** and a ***finish state***. Most tags in markup are accompanied by a closing tag. It's often the case, especially with markup that produces tables, that before a tag is closed, that another "opening" tag will appear. Another way of saying this is that markup languages allow tags to be nested within other tags. In order to create such markup, ODS needs to call the event that corresponds to the inner tag before it completes the instructions in the event that corresponds to the outer tag. For that reason, the outer event will be defined with a start state which contains instructions for text generation before nested events are called. The finish state then contains instructions for generating text after text has been generated for nested events. The syntax is as follows.

```
DEFINE EVENT ;
start:
   event statements

finish:
   event statements

END ;
```

Just as the data set variable describes a particular aspect of data, the event variable describes a particular aspect of the output delivery process. More specifically, the data set variable provides a unique piece of information about each observation of a data set, but the output delivery process has no counterpart to the observation. The aspect of the output delivery process that a given variable describes occurs at a particular moment in time (when a particular event occurs) in the process. When that moment or event has passed, the value of the variable is gone (though in some cases it may resurface in another event). For example, suppose you issue a TITLE statement in order to put a title at the beginning of your output. At least one of the events among those that ODS and the PROC use will now have access, through a variable, to the text that makes up this title. To get it into the markup in a place where the viewing program will interpret it as a title, the tagset must include in the definition of this event a PUT statement that writes this title to the file while the value of the variable contains it.

The data set variable, in a well-structured data set, typically provides the same information for each observation, but because the event variable is time-sensitive, describing only a piece of the process, several events may use the same variable for their own purposes. Never is this more true than with the event variable VALUE, one of the most common variables used. Let's go back to the TITLE example. In an event that is used early in the process, the value of VALUE is the text that makes up the title. Later in the process, the PROC event might use VALUE to hold the name of the PROC. Even later in the HEADER event, VALUE will hold the column header for the first column. At this point, the value of COLSTART, another event variable that, not being as flexible as VALUE, usually only serves one purpose, holds the value of 1. Still in HEADER, when COLSTART changes to 2, VALUE then holds the column header of the second column. We'll soon develop ways to know which events use which variables to hold information that is important for us.

ODS and the PROCs make use of over 500 variables to share information and generate text. Some can be classified as metadata while others are more directly related to the output. Many of the latter are populated by the PROC while others are populated by system options, ODS options, and other global statements. We've discussed VALUE and COLSTART – other common ones include COLCOUNT which, when captured at the right time, holds the number of columns in a table; EVENT_NAME, the name of the SAS event which, though not useful in output, will help us in our quest for information about events; COLSPAN and ROWSPAN which hold the number of columns/rows a cell is spanning. A full list of variables can be found in the Online Documentation.

Inside the event the user can create variables using the SET or EVAL statements. As a general rule, EVAL is used for numeric variables while SET is used for characters. EVAL can be used for mathematical and logical operations, while SET can be used to create character variables and arrays. Most user-created variables are classified as memory variables and are preceded by a $ both in the SET or EVAL statement that defines them as well as when referring to them. Stream variables, preceded by $$ when created and referenced, are also available, and are used to hold large amounts of data. The beauty of a user-created variable is that it retains its value until it is either explicitly re-initialized or it is deleted with the UNSET statement. This means that if necessary, we can capture the value of a variable when it has meaning to us, and hold onto it until we need it. Many of the DATA step functions and operators are available.

Other familiar DATA step concepts such as conditional statement execution, iterative execution, and statement blocks have a place in event definitions, albeit with a slightly different look. The syntax of a conditional statement has the condition placed at the end of the statement following a slash. While it can use familiar DATA step functions, more efficiency is realized from the use of tagset functions such as CMP whose two arguments are tested for equality; ANY and EXIST which return a value of TRUE when any of the variables listed in the argument are populated and all are populated, respectively; NOT which negates a condition and CONTAINS which looks for the second argument in the first argument. ELSE is allowed, but only within a statement block that begins with DO and ends with DONE. Iterative looping is achieved with the WHILE condition on the DO statement and works just like the DO-WHILE statement in the DATA step.

A variety of different PUT-type statements are also available. PUT works exactly as it does in the DATA step, with one exception that holds true for all flavors of PUT – when a variable name follows literal text (enclosed in quotes), if the variable has no value, the literal text preceding it will not be output. Other useful flavors of PUT are PUTQ which wraps quotation marks around the value of a variable being output, PUTLOG which directs output to the log file, and PUTVARS, which loops through all the variables in a variable group (e.g. memory variables, event variables, etc.) and with each iteration, populates the variable _NAME_ and _VALUE_ with the name and value of the current variable, respecively. PUTVARS is most useful when followed by a combination of literal text and _NAME_ and/or _VALUE_, which will generate output for each variable in the variable group.

Though many other event statements exist, other common ones include NDENT, XDENT, TRIGGER, and BREAK. NDENT has the effect of indenting any text that is output after the NDENT statement by an amount specified in the INDENT= statement outside of any event definition. XDENT has the opposite effect. BREAK stops and exits the event, ignoring subsequent statements, and is commonly executed conditionally. The argument of the TRIGGER statement is the name of another event, and has the effect of interrupting the current event to execute statements in the triggered event. It's important to know that while a triggered event is executing, the value of the event variable EVENT_NAME is still the name of the SAS event and not the triggered event. The variable TRIGGER_NAME holds the name of the triggered event. Also, if TRIGGER is executed during a particular state (e.g. the Start state), then only that state will be triggered from the triggered event, unless explicitly stated otherwise (e.g. `trigger otherevent finish`).

GETTING STARTED

Our toolbox is now off to a good start. We'll get into more details on those that we've discussed and any new ones as they become necessary, but we're now ready to begin building some tagsets of our own. The first tagset we'll start with can be thought of as another addition to our toolbox. After that we'll develop the tagsets that produced figures 2 through 4 above. Finally, we'll use these additional tools for modifying pieces of ODS output.

We'll begin by answering a question we posed earlier: what are the names of the events that ODS calls, and in what order are they called? To answer this, we'll make use of the event variable EVENT_NAME. For starters, as each event is called, all we want it to do is tell us what its name is. We can do this by writing a tagset with one event that writes out the name of the event, and set this as the default event.

```
proc template ;
define tagset info1 ;
default_event=”basic” ;
embedded_stylesheet=yes;

define event basic ;
    put “The name of this event is “ event_name nl ;
end ;

end ;
run ;
```

Recall that the default event is always used when ODS calls for an event that is not defined with the tagset. In this case, without a parent tagset, the BASIC event is always used sine no others are defined. Also note the use of the EMBEDDED_STYLESHEET statement. By setting this to Yes, we are assured that stylesheet markup is contained in the same file as the rest of the markup, as opposed to being directed to an external file or nowhere at all. In this case, we include it in order to see what style events are called. Finally, note the “nl” at the end of the PUT statement. This moves the pointer to the next line. We'll now use the tagset to create two different files.

```
(1) ods markup tagset=info1 file=”info1.txt” ;
    ods markup close ;

(2) ods markup tagset=info1 file=”info1a.txt” ;
    proc print noobs data=sashelp.class(obs=10) ;
    var name age sex height weight ;
```

```
run ;
ods markup close ;
```

Example (1) has the unusual feature that an ODS destination is being opened and closed with nothing happening in between. This allows us to know what events ODS calls, and in what order, before a PROC is ever processed. They are INITIALIZE, DOC, DOC_HEAD, DOC_META, AUTH_OPER, DOC_TITLE, and STYLESHEET_LINK. When no stylesheet is included, the next events are JAVASCRIPT, STARTUP_FUNCTION, SHUTDOWN_FUNCTION, and DOC_BODY. When a stylesheet is included as in our example here, between STYLESHEET_LINK and JAVASCRIPT is EMBEDDED_STYLESHEET, STYLES, many calls to STYLE_CLASS, and SHORTSTYLES.

When a PROC is included, you will see the same events as listed above at the beginning of your output, followed by the events the PROC uses. For the most part, all the PROCs use the same skeleton of events, but in some cases, slight differences do exist. For example, the REPORT procedure keeps column number information in the DATA event, but the cell content in the PUT_VALUE event immediately following DATA, whereas PROC PRINT, like many other PROCs, makes no use of PUT_VALUE, keeping column number information and cell content both in DATA. Typically, the first event after DOC_BODY (the last event before the PROC takes over) is the PROC event which, among other things, holds the name of the PROC. You'll see that the next set of events are "setup" events, setting up things like the title, the table, cell specs, row specs, table headers and others. We know the data is coming when we next get to events like HEADER, ROW, and DATA.

I've mentioned only a few of the events here – the list from top to bottom is fairly long. Often times the task of finding the event you're looking for and where it fits in with the others is just as challenging as identifying the event of interest in the first place. The best rule of thumb when trying to get a general idea of when events occur is to think of the tasks involved in building a table and the order in which they should be performed. First, the file needs to be built, then the PROC begins, titles are put in place, the table as a whole is set up, then the headers are filled in, rows are defined, and finally, column by column, data cells are filled in. For the most part, when trying to customize pieces of output, it is the later events that we have to work with. When changing the table itself, the events that follow the table setup are what we are interested in.

With some idea of what events are used and the general order in which they are used, let's now try and get an idea of when different types of information show up at what point in the process. We'll do this simply by adding a few variables to the above tagset.

```
proc template ;
define tagset info2 ;
default_event="basic" ;
embedded_stylesheet=yes;

define event basic ;
  start:
    put "EVENT_NAME:  " event_name nl ;
    put "HTMLCLASS:  " htmlclass nl ;
    put "COLCOUNT:  " colcount nl ;
    put "COLSTART:  " colstart nl ;
    put "COLSPAN:  " colspan nl ;
    put "ROW:  " row nl ;
    put "VALUE:  " value nl ;
    put "STATE:  " state nl ;
    put nl ;

  finish:
    put "EVENT_NAME:  " event_name nl ;
    put "HTMLCLASS:  " htmlclass nl ;
    put "COLCOUNT:  " colcount nl ;
    put "COLSTART:  " colstart nl ;
    put "COLSPAN:  " colspan nl ;
    put "ROW:  " row nl ;
    put "VALUE:  " value nl ;
    put "STATE:  " state nl ;
    put nl ;

end ;

end ;
run ;
```

With this tagset we've added a couple of new features. Maybe the one that stands out the most is the use of the Start state and Finish state. This will give us a good idea of when events are nested inside of each other. Also notice the "`put nl ;`" at the end of each state. This has the effect of inserting a blank line between events in the output. Figure 5 illustrates a snippet of PROC PRINT output, using the data set SASHELP.CLASS and the tagset above.

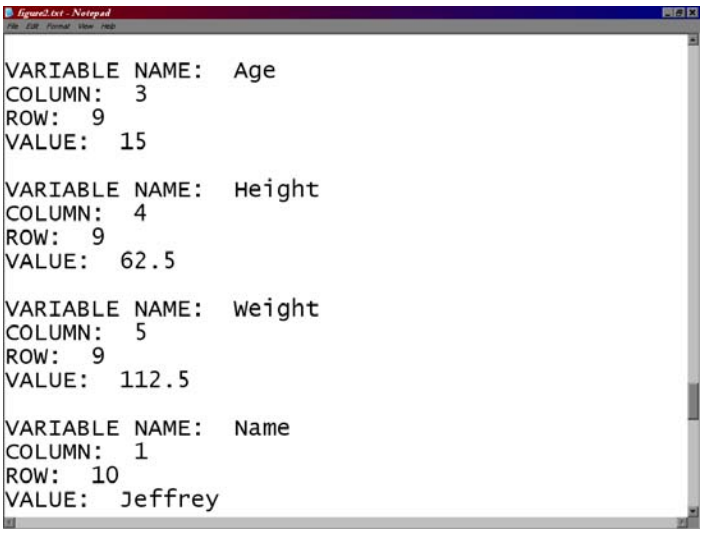


Figure 5

Let's see what kind of information some of these variables provide. Keep in mind that PUT statements in tagsets that combine literal text with variable names will only produce output for a text-variable combination when the variable has a value. If it doesn't, the text that precedes it will not be generated. For our purposes here this is a useful feature of the PUT statement. It means that for any given event, if, for example, we're missing the output COLCOUNT:, we know that this variable has no value in this event. EVENT_NAME always has a value, and because the BASIC event in this tagset was defined with states, STATE will always have a value too. Beware though, sometimes variables are populated with meaningless, or even worse, misleading information. COLCOUNT is an example of the latter. COLCOUNT claims to hold the number of columns in the table, but before the table is set up, this variable has a value of 1. The VAR statement in our PROC PRINT listed five variables, so the table should contain five columns, and later when the table begins to set up, COLCOUNT will have a value of 5. COLSPAN in this example never has a value. This variable does become useful in PROCs such as PROC REPORT that allow us to have spanning headers. As mentioned earlier, VALUE is a multi-purpose variable that holds the name of the PROC early, later the title from the TITLE statement, and even later, cell data values. ROW and COLSTART, as expected, hold row and column numbers respectively as the table is constructed. Finally, we have HTMLCLASS, which holds the name of the stylesheet class to be used. A detailed discussion on HTML style classes is beyond the scope of this paper (though it will be discussed more in an upcoming example), but by adding PUTVARS STYLE followed by a combination of _NAME_, _VALUE_, and possibly some literal text will provide the name (_NAME_) of each style variable and its value (_VALUE_) for each event. Adding this to a tagset similar to those above will generate output that resembles the style element defnitions in a style template.

Let's now return to Figures 2 through 4. These last couple of examples have illustrated how the output from Figure 2 can be generated. Though it has a much different appearance than that of Figure 2, Figure 3, upon close inspection, isn't much different. It contains a few more variables, is laid out in a more horizontal, XML format, and is viewed through Internet Explorer, but like the others, provides variable and value information about each event. Tagsets like this are often referred to as ***mapping tagsets***. SASHELP.TMPLMST contains several mapping tagsets, such as EVENT_MAP and SHORT_MAP. EVENT_MAP was used to generate Figure 3.

Unlike Figures 2 and 3, Figure 4 does not provide output for each event. Not only that, but this output contains no information about any events at all. Rather, it looks like nothing more than literal text, generated from a PUT statement in one event. By placing this statement in an early event, this can be generated without a PROC being run.

```
proc template ;
define tagset helloworld ;

define event initialize ;
put "HELLO WORLD" ;
end ;

end;
```

```
run;
```

THE FREQ EXAMPLE

We've now added an important tool to our toolbox – the mapping tagsets. This will prove to be valuable in identifying events of interest.

The FREQ procedure for the production of crosstabular output provides some opportunity for output customization. Throughout these examples we'll use the following code.

```
proc freq data=sashelp.class ;
title "Tagset example";
tables sex*age / missing norow nocol nopercent;
label sex='Gender' age='Age as of Dec. 31';
run;
```

Some quick observations are noteworthy here. First, note the use of labels for each of the variables being analyzed. Second, the combination of options in the TABLES statement ensures that only the frequencies will appear in the table cells. Our initial output generated from the SAS-supplied PHTML tagset (ODS PHTML file=…) is below.

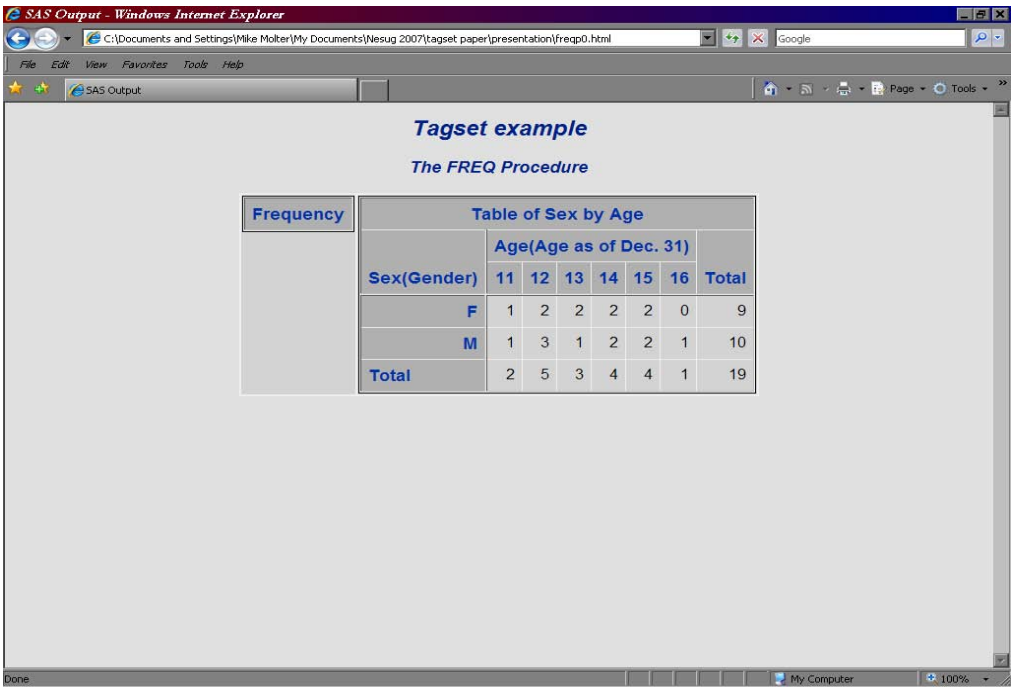


Figure 6

THE PROC TITLE

Our first task will be to do something with "The FREQ Procedure". We know that "Tagset example" came from a TITLE statement, but this text comes by default with FREQ output. If we're going to do something about it, we need to pinpoint which PHTML event was responsible for generating it. We first look to the event map. This is done by running the above code using the following ODS statement, or any of its equivalents.

```
ods tagsets.event_map file="file name.xml" ;
```

Below is a small piece of the XML output.

```
- <proc_title_group event_name="proc_title_group" trigger_name="attr_out" class="Body"
    colcount="1" index="IDX" just="c">
  <proc_title event_name="proc_title" trigger_name="attr_out" class="ProcTitle" value="The
    FREQ Procedure" colcount="1" index="IDX" just="c" />
   </proc_title_group>
- <proc_branch event_name="proc_branch" trigger_name="attr_out" class="ContentProcName"
    value="Freq" colcount="1" name="Freq" label="The Freq Procedure" index="IDX" just="c"
    url="freqem0.xml#IDX" hreftarget="body">
```

In this piece, we see that “The Freq Procedure” is the value of the VALUE variable in the PROC_TITLE event, but also the LABEL variable in the PROC_BRANCH event. At this point, either of the events could have generated the output. We need to investigate further.

Another good resource is the markup itself. By opening the output with a plain text editor such as Notepad, we discover the following contained within the file.

```
<noscript></noscript>
<div class="branch">
<a name="IDX"></a>
<h1 class="c">Tagset example</h1>
<h2 class="c">The FREQ Procedure</h2>
<p>
<div>
<div align="center">
```

From this, we see that the text is contained within the h2 tags. Whichever of the two events were responsible, it probably also generated this tag. We now look at the PHTML tagset code, do a simple search for “<h2”, and discover the following:

```
define event proc_title;
    put "<h2";

    trigger align;
    put ">";
    put VALUE;
    put "</h2>" NL;
 end;
```

We don’t see the “The FREQ Procedure” as literal text in this event, but all the clues point to this text as being the value of the VALUE variable. Assuming this, we can do what we want. We can inherit the tagset and re-define the event as an empty one, as below, thereby suppressing the text.

```
proc template ;
define tagset freq1 ;
parent=tagsets.phtml ;

define event proc_title;
end;

end;
run;
```

We can also substitute another variable for VALUE, substitute literal text such as “The FREAKY Procedure”, or we could create a memory variable in the event and replace VALUE with a reference to it, as in the following.

```
proc template ;
define tagset freq1 ;

define event proc_title;
set $myvalue “THE FREAKY PROCEDURE” ;
put "<h2";

trigger align;
put ">";
put $myvalue ;
put "</h2>" NL;

end;
end;
run;
```

Not all PROCs use the PROC_TITLE event, and for those that do, the ODS PROCTITLE global statement can be turned on and off to control whether this kind of text is displayed or not. In reality, rather than using a tagset to customize this text, one might choose to turn off its generation and substitute it with another title. The value in this

example is more in the lessons we've learned than in the usefulness of this feature. We've seen examples of the tagset structure, the event structure, how statements are used, how events call each other, and the syntax. Maybe more importantly, we've gained experience as a detective. We learned about different resources available to pinpoint the responsible event. This was done by a combination of mapping tagsets (either EVENT_MAP, SHORT_MAP, or a homemade tagset), examination of the markup itself, and simple searching through the parent tagset. Finally, we've made more sense of an ODS feature that in the past, was another example of ODS magic. Issue a statement and somehow this text would appear in, or disappear from the markup. We now know that the text "The FREQ Procedure" is just one example of a piece of information internal to this PROC, and passed, through an event variable, to ODS for output delivery when this statement is turned on. By turning the statement off (ODS PROCTITLE=OFF) and running the PROC through the event map, we would see that the PROC_TITLE event is never used. As is the theme with tagsets in general, we've gained a better glimpse into the inner workings of this PROC, and we've gained control over a piece of the markup that was at one time untouchable.

"SAS OUTPUT"

If you look carefully at the upper left corner of the browser window in Figure 6 above, you will see the text "SAS Output." If you're using Internet Explorer version 7, you will see the same text in the tab (as is the case in the figure above). Let's see if the investigative techniques we used above will help us to get to this text. Unfortunately in this case, we don't find this text in the event map XML file. Maybe this is something specific to HTML. Looking at the markup, we find the following.

<title>SAS Output</title>

A search for "<title>" in the tagset code turns up three possible events – TOP_TITLE, DOC_TITLE, and CONTENT_TITLE. What's different about this example is that the text we're trying to get to, in each of these possibilities, appears to be written out as literal text. In each case, the text is being generated under the condition that VALUE doesn't exist. If it does, then its value will be generated. Although it's not definitive evidence, TOP_TITLE can probably be ruled out, because the literal text it is putting out is "SAS Output Frame". It's possible that VALUE=SAS Output, in which case, this event could be responsible, but more likely it is one of the other two, each of which would generate "SAS Output" when VALUE doesn't exist. We can always return to this other possibility if these two events prove innocent. By returning to the event map output, though the text is never found, when searching for CONTENT_TITLE, we discover that this event was never used, leaving DOC_TITLE as our leading suspect. We test our suspicion by inheriting the PHTML tagset or even the FREQ1 tagset from above, and customizing this event the way we did with the PROC_TITLE event. Sure enough, DOC_TITLE turns out to be the guilty party.

SUPPRESSING ROW AND COLUMN VARIABLE NAMES

When the row and/or column variables have labels attached to them, PROC FREQ tends to display in the table the name of the variable followed by the label inside parentheses. Let's see if we can get rid of the variable names and parentheses, leaving only the labels. Because this text plays the role of a header, this might be a good place to start looking in the event map. Unfortunately, we won't like what we find. The generation of the row header is much different than that of the column header, forcing us to treat them separately.

We'll begin with the row header. An excerpt from the event map that contains it is below. Pay close attention to the values of VALUE.

```
- <header event_name="header" trigger_name="attr_out" output_name="CrossTabFreqs"
     output_label="Cross-Tabular Freq Table" section="head" class="Header" value="Sex"
     rowspan="2" row="2" data_row="0" colcount="1" col="1" type="string" index="IDX" just="c"
     vjust="b">
   <put_value event_name="put_value" trigger_name="attr_out" output_name="CrossTabFreqs"
     output_label="Cross-Tabular Freq Table" class="Header" value="(" colcount="1"
     index="IDX" just="c" vjust="b" />
   <put_value event_name="put_value" trigger_name="attr_out" output_name="CrossTabFreqs"
     output_label="Cross-Tabular Freq Table" class="Header" value="Gender" colcount="1"
     index="IDX" just="c" vjust="b" />
   <put_value event_name="put_value" trigger_name="attr_out" output_name="CrossTabFreqs"
     output_label="Cross-Tabular Freq Table" class="Header" value=")" colcount="1"
     index="IDX" just="c" vjust="b" />
```

What is disturbing is that the text "Sex(Gender)" is split across four event uses, with the variable name Sex in the HEADER event, and the rest in separate uses of the PUT_VALUE event. Let's start with the HEADER event. When looking at this event definition in the PHTML tagset definition, you'll see that it triggers another event, CELL_VALUE, which is what is responsible for the generation of the output. Here you'll find VALUE being generated, under the condition that the event variable URL doesn't exist. That's only because when that variable does exist, VALUE will be output in another event that includes text to create a hyperlink. Because we don't want to generate the variable

name, we have another instance to suppress output generation. Now because HEADER (and in turn, CELL_VALUE) is responsible for several parts of the output, we have to place tight conditions on suppressing this particular variable name. One condition would be to insist that the row number is 2. Row 1 will always contain the label of the table (in this case, "Table of Sex by Age". Of course we don't want to suppress everything in row 2, so we need more conditions. We also know that this label will span two rows, the first of which contains column headers, the second that contains column variable values. We're getting closer, but the Total column also spans two rows. A third condition could be that the column number is 1. This should uniquely identify the text to suppress. The PUT statement in the CELL_VALUE event is now modified to contain extra conditions, as in the following.

```
put value /if ^exists( URL) and ^(cmp("2",row) and cmp('2',rowspan) and
cmp("1",colstart));
```

EXISTS is a function that returns true when all the variables in the list have values. The carrot (^) negates the result of any function, and CMP checks for equality among its two arguments. ROW and COLSTART hold the row and column number of the current cell respectively, and ROWSPAN holds the number of rows spanned by the current cell.

That takes care of the variable name – now it's time for the parentheses. By looking again at the event map, it turns out that this is the only cell that calls on the PUT_VALUE event. By looking at the PHTML tagset definition, we see that all this event does is write out the value of VALUE. We now simply add the condition that this value not be an opening or closing parenthesis.

```
define event put_value;
   put VALUE / if value not in ("(",")");
end;
```

We've now trimmed down "Sex(Gender)" to "Gender". We now want to reduce "Age(Age as of Dec. 31)" to "Age as of Dec. 31". The good news is that this is found all together from one event in the event map.

```
<header event_name="header" trigger_name="attr_out" output_name="CrossTabFreqs"
output_label="Cross-Tabular Freq Table" section="head" class="Header" value="Age(Age as of
Dec. 31)" colspan="6" row="2" data_row="0" colcount="1" col="2" type="string" index="IDX"
just="c" vjust="b" />
```

The bad news (ok, it's not that bad) is that it's in the HEADER event. We will again have to impose tight restrictions to make sure this is the only cell we affect. Though many combinations would work, we should be able to count on ROW=2 and ROWSPAN not existing. The question now is what to do under these conditions.

Recall that HEADER leaves it up to CELL_VALUE to generate the text – specifically, VALUE. Under these conditions, we want it to generate just a portion of VALUE. For that reason, we will use DATA step functions in HEADER to parse VALUE, storing the result in a memory variable. In CELL_VALUE, we will output the value of this variable when it exists. When it doesn't, we'll generate what we did before under the conditions that were added above.

It's important to remember here that HEADER is still passing something to CELL_VALUE for generation. That means we don't want to remove anything from the current definition of HEADER – we still want to generate the tags and trigger the same events. We just want to have our new value ready before CELL_VALUE is triggered.

It's also important to know that when parsing a string, tagsets don't react very well to nested functions, so a few more lines of code may be necessary. We begin by finding the occurrence of the first open parenthesis.

```
do / if cmp("2",row) and ^exist(rowspan);
eval $pos index(value,"(") ;
```

Notice that we have begun a statement block with DO under the conditions stated above. We then used EVAL to create a numeric variable with the familiar INDEX function. Since we want nothing before this open parenthesis, or either of the parentheses, we will use SUBSTR starting one position after the result of our INDEX function.

```
eval $pos $pos+1 ;
set $val substr(value,$pos) ;
```

This is nice, but we still have a closing parenthesis at the end. We can chop that off with some help from the LENGTH function.

```
eval $len length($val) ;
eval $len $len – 1 ;
```

```
set $val substr($val,1,$len) ;
unset $len ; unset $pos ;
done ;
```

This statement block has been added to the HEADER definition immediately before the call to the CELL_VALUE event (TRIGGER CELL_VALUE). We now alter CELL_VALUE to generate this when it's available.

```
do / if $val ;
    put $val ;
    unset $val ;
else ;
    put value /if ^exists( URL) and ^(cmp(row,"2") and cmp('2',rowspan) and
    cmp("1",colstart));
done;
```

Under the ELSE statement is what we came up with earlier. The condition in the DO statement that appears not to have an operator works like an EXISTS function. Maybe the most important part of this is the UNSET statement. Keep in mind that memory variables hang around forever until you explicitly do something with them. Without UNSET, $VAL will persist and its value generated whenever this event is called.

SUPPRESSING THE "MINI" TABLE

The crosstabular PROC FREQ generates two tables – of course the main table with all the frequencies in it, and the second is the one-by-one table to the left that tells you what each number in any cell represents. Having suppressed all but the frequencies, let's see if we can suppress the generation of this mini table.

Suppressing an entire table means suppressing a bigger chunk of markup, starting with markup that begins with the text "<table". A little investigation reveals that the TABLE event is responsible for this. The temptation at this point might be to look for something in the event map that distinguishes this table from the main table. Unfortunately, the variables that EVENT_MAP shows us don't show any difference. You could write your own mapping tagset to look for such a variable, but that can take a while, and in the end, probably is not worth the effort. The reason is because the TABLE event has several events nested within. Whatever condition you find will have to be imposed on all those events so that they too don't generate any output. Rather, we'll make use of the stream variable. With this approach, we minimize the amount of modification to existing code, and add one new, short event.

Streams can be initialized with the SET statement or the OPEN statement. When we initialize a stream variable with an OPEN statement, output from all PUT statements is redirected to the stream instead of to the file, until the stream is closed. For that reason, with all the PUT statements contained in the TABLE event, we don't have to change anything in the current TABLE event, as long as we make sure that when this is called for the mini table, a stream is open, and when for the main table, no stream is open. What we will do to the current TABLE event is rename it, say, to MYTABLE. We will then redefine TABLE to manage the opening and closing of the stream. TABLE will then trigger MYTABLE. The code is below.

```
define event table ;
start:
open dump / if ^exist($is_table) ;
trigger mytable ;

finish:
trigger mytable ;
close ;
set $is_table "true" ;
end;
```

The memory variable $IS_TABLE manages the stream. Since the mini table is generated first, we want to open the stream at the first use of the TABLE event. Since $IS_TABLE has yet to be initialized, the condition for opening DUMP is true at this point. Now all the markup generated by MYTABLE (normally TABLE) that creates a table, as well as everything generated by nested events before the FINISH state, is directed to the stream. When we finally reach the FINISH state, we trigger the FINISH state of MYTABLE which generates the closing table tag (</table>), finalizing the creation of a table. We can now close the stream and initialize $IS_TABLE, so that the stream is never open again, and all subsequent PUT statements direct text to the file.

PHTML VS. HTML – A MATTER OF STYLE

The main difference between these two ODS destinations is in styling. Once again we encounter an aspect of ODS output that once was a mystery, but now becomes clearer with the tagset. We know that we can change the way our output looks by specifying the name of a style definition with the STYLE= option on the ODS statement. Furthermore, we can use style definitions provided by SAS, or we can create our own by setting attributes for each of the different

style elements. Some PROCs also allow us to define style attributes “on the fly” as a way to temporarily override the attribute values defined on the style template being used. TITLE and FOOTNOTE statements also allow for this kind of manual override, or what is also referred to as inline styling. Regardless of the method used, the question still remains: how do these specifications get into the markup? We’re now in a better position to answer that question.

We saw earlier after turning on the EMBEDDED_STYLESHEETS statement which style events ODS uses. The main event is called STYLE_CLASS. Each time this event is called, another element from the style definition is loaded – the name of the element (defined in the style definition) becomes the value of the variable HTMLCLASS, and all of the attribute values become variable values, all accessible in the tagset (e.g. if BORDERCOLOR is defined for the DATA element as green, then when STYLE_CLASS loads this element, HTMLCLASS=DATA while BORDERCOLOR=GREEN). With these variables populated, simple PUT statements create valid stylesheet syntax, either at the top of the markup file when EMBEDDED_STYLESHEETS is turned on, or in an external file when one is specified on the ODS statement. PHTML, however does not define the STYLE_CLASS event. Rather, PHTML uses a more manual method of creating its stylesheet with the SHORTSTYLES event. Instead of loading each of the style elements in the definition, this event triggers a small number of events, each of which specifies one element to load. As with STYLE_CLASS, variables become populated and, PUT statements create a much smaller stylesheet.

Though the stylesheet PHTML generates is quite a bit shorter than that from the HTML destination, this difference isn’t always seen by the user. The reason is that the few elements that are loaded are the ones users care about most of the time – BODY, SYSTEMTITLE, PROCTITLE, TABLE, DATA, HEADER. Therefore changes to these elements in the style definition will be realized by the PHTML destination. A much more visible difference between the two destinations is the inability of PHTML to handle inline styling.

Without knowledge of tagsets, no logical explanation can be offered for how we can change the font color of the title by changing the Systemtitle element definition in the style definition, but we can’t change it with the statement

```
title color=red “Why cant I change title colors?” ;.
```

As mentioned above, Systemtitle is loaded by the PHTML destination. Inline style attributes, however, never become part of an element definition. Their values still populate variables, but they do so during the event being used to generate that part of the output – much later than when STYLE_CLASS and SHORTSTYLES are used. When the above statement is issued with the HTML destination, the value of FOREGROUND is populated when the SYSTEM_TITLE event is used. This ultimately calls the STYLE_INLINE event that generates inline style attributes inside the tag that produces the title. The DATA event does the same thing. Oddly enough, though the STYLE_INLINE event that the HTML destination calls is actually defined in the PHTML tagset (HTML tagset inherits from PHTML), the SYSTEM_TITLE and DATA events defined in PHTML never call STYLE_INLINE.

THE BY STATEMENT

Another noticable difference is seen when a BY statement is used with a PROC. When this happens the BYLINE event is used. The PHTML and HTML destinations define this event differently. The following illustrates the differences.

PHTML output:

```
<h1 class="c">Sex=F</h1>
```

HTML output:

```
<div class="c Byline">Sex=F</div>
```

The CLASS= attribute inside an HTML tag tells the browser where to look in the stylesheet to get style instructions. The stylesheets in both destinations contain the text .c {text-align: center }; CLASS=”c” means look at this part of the stylesheet for instructions. In this case, the instruction is to center align the text. With PHTML, text-alignment values such as these are the only values provided in this attribute. Without any other style instructions, the browser looks to the element in the stylesheet named for the tag itself – h1. As it turns out, PHTML puts all of its titles in h1 tags too. For that reason, By lines in this destination look just like titles.

On the other hand, with the HTML destination, the browser has one more place in the stylesheet to look for instructions – the Byline class. This class defines the same attributes as the h1 class with different values, giving the By line its own unique ook.

```
.Byline
{  font-family: Arial, Helvetica, sans-serif;
  font-size: medium;
  font-weight: bold;
  font-style: normal;
  color: #0033AA;
  background-color: #B0B0B0; }
```

The dot in front of the class name means that any element can reference this class inside the tag. The STYLE_CLASS event loads each element defined by the style definition and creates classes such as this one in the stylesheet, named for the element. The SHORTSTYLES event used by PHTML also creates classes named for the few elements it loads, but omits the dot, meaning that references to the class are permitted only inside the tag with the same name. Additionally, each destination adds classes for text alignment. Since Byline is not one of the elements that PHTML loads, it does not get added as a class to its stylesheet, and there is no opportunity to reference it in any of the tags. For that reason, Byline attributes have to be borrowed from one of the few classes it does have. Let's see if we can create a tagset inherited from PHTML that will load the Byline element and add it as a class option to the above output.

The SHORTSTYLES event in PHTML does nothing more than trigger several events, each of which has the same basic structure illustrated below.

```
define event titlestyle;
put "h1 {" NL;
trigger stylesheetclass;
put "}" NL;
style = systemtitle;
end;
```

The PUT statement in this example is beginning to generate the class that only h1 tags can use. The STYLESHEETCLASS event will generate a few attributes (e.g. font face, font size, font weight, plus a few others) and their values retrieved from variables. STYLE=SYSTEMTITLE means that these variables are populated with values defined by the SYSTEMTITLE style element.

Following this convention, we inherit the PHTML tagset and add a new event called BYSTYLE.

```
define event bystyle ;
put ".byline {" NL;
trigger stylesheetclass;
put "}" NL;
style = byline;
end;
```

Note that we've added the dot in front of the class name so that we can reference it from anywhere. We are also loading the style element BYLINE, the same one loaded by the HTML destination that gives this part of the output its own unique look. We can now add this event as one to be triggered by the SHORTSTYLES event. All that remains is adding the reference alongside the "c" in the text <h1 class="c">. After generating "<h1", the current PHTML definition triggers the event ALIGN, which generates the CLASS= attribute above, including the closing quotation marks following the c. By changing the event triggered to CLASSALIGN, before the closing quote gets generated, the value of HTMLCLASS (in this case, BYLINE) gets generated. The new event looks like the following.

```
define event byline;
put "<h1";
trigger classalign;
put ">";
put VALUE;
put "</h1>" NL;
end;
```

The attribute inside the h1 tag now reads class="c byline", which tells the browser to look to both the "c" class and the "byline" class for styling instructions.

TAGSET PARAMETERS

Version 9.1.3 introduced the ability to parameterize our tagsets similar to the way we parameterize macros, with two main differences. While macro parameters are declared on the %macro statement, tagset parameters are never declared. The second difference is that while each macro parameter value is stored in its own macro variable, each tagset parameter is stored in the same dictionary variable. A dictionary variable is one of two ways we can create arrays in the tagset. Before moving on, let's look at this capability in a little more detail.

TAGSET ARRAYS

In addition to the variables we've seen up to this point, the SET statement can also create list and dictionary variables, each of which works like an array. Of the two, the list variable is more like the familiar DATA step array. In addition to the name of the variable, we add a set of square brackets and an optional index, as in the following.

```
set $mylist[1] 'first element' ;
```

The index specifies which particular element in the list will hold this value. Providing two square brackets without an index in between (i.e. `set $mylist[] 'last element'`) adds another element to the end of the list to hold this value. References to the variable are the same as references to a DATA step array element.

```
put 'The 24th element of this list variable is ' $mylist[24] ;
```

The only difference that dictionary variables offer is that instead of associating element values with integers, they are associated with other character strings.

```
set $states['Michigan'] 'MI' ;
set $states['North Carolina'] 'NC' ;
```

Note the use of quotation marks surrounding the character index inside the brackets. Referencing is the same as with list variables, but again with quotation marks.

```
put 'The abbreviation for Michigan is ' $states['Michigan'] ;
```

As an alternative to integer and string indeces, arrays can also use the name of another variable as an index, as in the following.

```
eval $loop 1 ;
do / while $loop < $mylist ;
    put "The current value of the MYLIST variable is " $mylist[$loop] nl ;
    eval $loop $loop + 1 ;
done ;
```

Other differences from the DATA step array exist too. Note that with both types of tagset arrays, array elements can be added at any time. This differs from the DATA step array where the ARRAY statement serves the purpose of declaring ahead of time exactly how many elements it will hold. Also, the DATA step has the DIM function whose argument is the name of the array, and returns the number of elements in the array. This information is available for list and dictionary variables by referencing the name of the variable without an index or brackets.

```
put 'The number of elements in the variable $states is '  $states ;
```

THE $OPTIONS DICTIONARY

Each tagset parameter becomes the name of an individual element in a dictionary variable called OPTIONS. Parameter values are provided by the user on the ODS statement that opens the destination in the following manner.

```
ods markup tagset=tagset-name options(parameter-name-1 = "parameter-value-1",…
parameter-name-n = "parameter-value-n") ;
```

We reference these parameter values in the tagset the way we reference any other dictionary elements, except for the following: regardless of how the user refers to the parameter name in the ODS statement, *references to the parameter names in the tagset definition must be in all caps.*

```
proc template ;
define tagset caps ;
define event initialize ;
putlog "The value of the parameter is" $options["PARM"] ;
end;
end;
run;

ods markup file='test.txt' tagset=caps options(parm="hello world");
ods markup close;
```

Note the reference to the dictionary element on the PUTLOG statement without ever having defined a variable. The user, rather than the author of the tagset, becomes responsible for defining the $OPTIONS dictionary variable with the OPTIONS option and its attributes.

The Proctitle and SAS Output examples illustrated earlier are begging for parameterization. In the Proctitle example, multiple options were suggested. The one illustrated had us creating a memory variable with the PROC_TITLE event, and referencing it in a PUT statement in the same event. Though this does allow for customizing this part of

the output, it's about as desirable as asking a user to modify macro code. This example becomes much more user friendly by rewriting the event as below.

```
proc template ;
define tagset freq1 ;

define event proc_title;
/* set $myvalue "THE FREAKY PROCEDURE" ; */
put "<h2";

trigger align;
put ">";
put $options["PTITLE"] ;
put "</h2>" NL;

end;
end;
run;
```

The user supplies a parameter value in the following manner.

```
ods markup file='myproctitle.html' tagset=freq1 options(ptitle="THE FREAKY
PROCEDURE");
```

We can make a similar change to the DOC_TITLE event so that "SAS Output" becomes what we want it to be. We change this event…

```
define event doc_title;
put "<title>";
put "SAS Output" /if ^exists( VALUE);
put VALUE;
put "</title>" NL;
end;
```

to this event

```
define event doc_title;
put "<title>";
put ' " ' $options["DOCTITLE"] ' " ';
put "</title>" NL;
end;
```

where DOCTITLE is the name of the parameter.

Let's go back to the styling of the By line. In this example we created a tagset that added a class to the stylesheet and added a reference to that class in the h1 tag. By adding `style=byline` to the new Bystyle event, we loaded the Bystyle style element and its unique attributes. Let's now parameterize that tagset to allow for a user to have control over some of those attributes.

Recall that adding the class reference to the h1 tag was done by modifying the BYLINE event. We can leave that part alone. In order to change attributes, we need to change the class definition itself in the stylesheet. This was controlled by the new BYSTYLE event, and more specifically, the STYLESHEETCLASS event that it triggered. STYLESHEETCLASS is responsible for generating attribute-value output with tagset statement such as the following.

```
put "  font-family: " FONT_FACE;
put ";" NL / if exists( FONT_FACE);
```

The value of FONT_FACE comes from the style element loaded (BYLINE). A simple change to this produces the font face chosen by the user.

```
put "  font-family:  " $options["BYFONT_FACE"] ;
put "  font-family: " FONT_FACE / if ^exists($options["BYFONT_FACE"]);
put ";" NL / if any($options["BYFONT_FACE"], FONT_FACE);
```

The first statement simply replaces the variable inherited from the loaded style element with the parameter input, where BYFONT_FACE is the name of the parameter. In case this wasn't specified by the user, we'll go back to

generating the font face loaded with the style element. The third statement, similar to the second in the original, generates the semicolon if either of the variables has a value. STYLESHEETCLASS is filled with pairs of statements like these, one for each attribute to be generated. It's certainly not much of a stretch to parameterize other attributes in the same way. If you're willing to go a bit further, do the same for other areas of the output, such as the titles, the Proctitle, table headers and even data. Depending on how many areas you do decide to affect, you may choose to make these changes directly to the STYLESHEETCLASS event, or create a separate event to be triggered, or if it's just one area of the output you want to parameterize such as the Byline, replace the trigger STYLESHEETCLASS statement with statements like these.

Of course an alternative way to gain the same functionality would be to change the style template. Templates, though, are in some ways like macros – we like to keep them relatively stable by avoiding making less-than-fundamental changes to them. Parameterization is a concept designed just for these kinds of changes. Inline styling offers the same convenience but we saw earlier how this is not accessible to PHTML (though an alternative to this might be a tagset that makes inline styling accessible). Either way, we avoid the sometimes difficult task of defining style definitions, and the style definitions that we do have each maintain their inidividual identities that makes each of them unique. All of this is accomplished by directly controlling the markup that ODS is generating.

THE REPORT EXAMPLE

Our final example re-emphasizes a theme touched on at the end of the last example. Without that tagset, if a user suddenly decides he wants the By line to have Times New Roman font, he is going to have to do some prep work before running the PROC – namely, creating a new style template that is different from the style template he was going to use in only a trivial way – the font of the By line. By creating a parameterized tagset, this prep work was eliminated, and the price (other than the initial tagset development) was a simple addition of an option (for specifying the parameter value) on the ODS statement.

PROC REPORT is one of the most versatile reporting PROCs we have. It's capable of generating not only a detailed report of your data, but also a summary report with all the basic descriptive statistics SAS has to offer, customized summary lines, and also custom inline styling. For these reasons, it's also one of the most widely used PROCs.

What the PROC is capable of though is one thing, what clients want is sometimes another. It's common for clients to request for certain pairs of descriptive statistics to be combined into one column. PROC REPORT is perfectly capable of calculating the mean and standard deviation, but there's no easy way to get them into one column with the standard deviation inside parentheses, or combine the minimum and maximum with a dash in between. For that reason we often find ourselves, as in the last example, preparing the data ahead of time, this time by computing the statistics that PROC REPORT is capable of computing ahead of time (maybe with the MEANS procedure) and then using PROC REPORT for display purposes only. In our final example, we'll eliminate that prep work by creating a parameterized tagset that combines two columns according to a pattern.

In this example, we'll use the following basic PROC REPORT.

```
proc report nowindows data=sashelp.class split='^' ;
column sex age=meanage age=stdage ;
define sex / group 'Gender' ;
define meanage / mean 'Mean^Age' format=8.1;
define stdage / std 'Standard^Deviation' format=8.2;
run;
```

This code creates a three-column report in which the second column represents the mean age for the current value of SEX and the third column represents the standard deviation of the age. The goal of the tagset is to allow the user to ask for the second and third column to be combined into one column (thereby making it a two-column report), with the standard deviation being wrapped in parentheses. The parameter would be specified in the following way.

```
ods markup tagset=squeeze file='squeeze sample.html' options(pattern="{2} ({3})");
```

The numbers within the braces represent column numbers calculated by PROC REPORT – in other words, as they would be with a "default" display of the report. In this case, we're saying that columns 2 and 3 of the default report (the mean and standard deviation) should be combined into one column by putting a space between them and the value of the third column in parentheses.

Though not always necessary, it's a good idea, when possible, to have in mind how you expect the markup to be changed. By running the above code with the PHTML tagset, the markup contains the following.

```
<colgroup>
<col>
<col>
<col>
```

```
</colgroup>
<thead>
<tr>
<th scope="col" class="c">Gender</th>
<th scope="col" class="c">Mean<br>Age</th>
<th scope="col" class="c">Standard<br>Deviation</th>
</tr>
</thead>
<tbody>
<tr>
<td class="l">F</td>
<td class="r">    13.2</td>
<td class="r">    1.39</td>
</tr>
<tr>
<td class="l">M</td>
<td class="r">    13.4</td>
<td class="r">    1.65</td>
</tr>
```

Since we're reducing the table to two columns, we should eliminate one of the <col>s. We'll also eliminate the second <th tag and the second <td tag within each <tr> tag, but in each case, we want to hold onto cell text (the value of VALUE, found between the opening and closing tags – e.g. Gender) so we combine it with its counterpart in the third tag. For example, the following will replace the first set of <td tags.

```
<tr>
<td class="l">F</td>
<td class="r">    13.2 (    1.39)</td>
</tr>
```

The following list of tasks describes the general plan of attack when it comes to writing the tagset.

- Parse the parameter. The user can specify any characters to separate the values from the two columns (parentheses, dashes, commas, etc.). We will find it handy to have a nice comma delimited list of the columns affected (column numbers still in braces). At the same time, we'll also want to know which of the two columns comes before the other.
- When processing the first of the two columns, we want to do two things – hold onto its value, but also suppress any output generation. We'll use a memory variable to accomplish the first, and we'll open a stream variable to accomplish the second.
- When processing the second of the two columns, rather than generating the value of VALUE, we'll create a memory variable simply by using the DATA step's TRANWRD function, replacing the number corresponding to the first column in $PATTERN with the value held from when that column was processed, and replacing the number corresponding to the current column with the current value of VALUE.

The first of these tasks can be done any time before the beginning of table creation (e.g. DOC event). A sample of the logic follows.

```
   set $numbers compress($options["PATTERN"],"1234567890{}","k") ;
   set $numbers tranwrd($numbers,"}{","},{");
   set $temp1 scan($numbers,1,",") ;
   set $ctemp1 compress($temp1,"{}");
   eval $ntemp1 inputn($ctemp1,'12.');
```

The first statement begins the creation of a memory variable called $NUMBERS that simply removes from $PATTERN everything but numeric digits and braces. The second statement inserts a comma between each set of braces. This gives us the comma delimited list of affected columns, which makes it easy to use the SCAN function to process each specified column one at a time. After stripping away the braces to create $CTEMP1, the last statement is used to turn the column number into a number. Subsequent statements do the same with the second column, and logic is used to determine which is bigger. Whichever of $TEMP1 and $TEMP2 holds the largest column number gets stored in $CSECOND, and the smaller in $CFIRST. $SECOND and $FIRST hold the numeric counterparts.

Two new events are added. One is called SUPPRESS and is defined as below.

```
define event suppress ;
   start:
   open dump  / if cmp(colstart,$cfirst) ;
```

```
   set $col colstart ;

   finish:
   close ;
   flush ;
end;
```

As you can see, the start state opens a stream called DUMP when the current column number (COLSTART) is the first column specified in the parameter ($CFIRST) and the finish state closes and flushes it. The current HEADER and DATA events are modified simply by triggering this event at the beginning of the start state and at the end of the finish state. This saves us from having to add conditions to all the PUT statements. Note also that the memory variable $COL is created to hold the value of the column number. The reason for this will become clear soon.

The second new event is called SETVALS, and serves the purpose of creating the necessary memory variables when either of the two columns specified in the parameter are being processed. The event is defined as below.

```
define event setvals ;
do / if cmp($col,$cfirst) ;
       set $hold value ;
else / if cmp($col,$csecond) ;
       set $val tranwrd($options["PATTERN"],$first,$hold) ;
       putlog "$VAL:  " $val ;
       set $val tranwrd($val,$second,value);
       putlog "$VAL:  " $val ;
done;
end;
```

A memory variable called $HOLD is created when the first of the two column is being processed, simply to hold onto that value. When the second column is being processed, $VAL is created – first by substituting the held value of the first column into $PATTERN, and second, by substituting the current value of VALUE in for the second column. The only question that remains is how and when this event is used.

The header of the table is generated in the same way other PROCs generate it. – by calling the HEADER event. In PHTML, this event calls the CELL_VALUE event which contains the PUT statement that generates the output. On the other hand, unlike other PROCs, the DATA event does not generate output. It contains the column number (COLSTART) but does not contain VALUE. Instead, PUT_VALUE, an event that contains nothing but the PUT statement, generates the output (this is the reason we created the memory variable $COL to hold onto the column number). Yes, the DATA event calls CELL_VALUE which has the PUT VALUE statement, but since VALUE is empty, this statement has no effect. Data cells have to wait until PUT_VALUE is called to have output generated.

In our case, whether its header or data output, we don't necessarily want VALUE being output, depending on whether it's a column in the parameter specification or not. So here's what we do. The PUT statement in the current CELL_VALUE event is executed if the variable URL is populated. We replace this conditional PUT statement with a TRIGGER statement, triggering a new and improved PUT_VALUE event that triggers SETVALS, and then generates $VAL if it exists, VALUE if it doesn't.

```
   define event cell_value;
   start:
   trigger preformatted /if asis;
   set $close_hyperlink "true" /if exists( URL);
   trigger hyperlink /if exists( URL);
   trigger put_value /if ^exists( URL) and ^cmp("DATA",EVENT_NAME);

   finish:
   trigger hyperlink /if exists( $close_hyperlink);
   unset $close_hyperlink;
   trigger preformatted /if asis;
   end;

   define event put_value;
   trigger setvals ;
   do / if exists($val);
   put $val ;
   unset $val ;
   else ;
   put value ;
```

```
done ;
end;
```

Keep in mind that the HEADER event contains a populated VALUE. When called, it triggers CELL_VALUE which then triggers PUT_VALUE. After checking with SETVALS to see if the current column is one that was specified with the parameter, it either writes $VAL or VALUE to the file. On the other hand, DATA does not contain a populated VALUE. For that reason, it does no good to call the PUT_VALUE and SETVALS event, which is why the TRIGGER statement in CELL_VALUE has the added condition that the name of the event is not DATA. We do know that, unlike other PROCs, REPORT calls PUT_VALUE on its own so that EVENT_NAME=PUT_VALUE. This is when the statements of PUT_VALUE execute for data cells.

These are the basic elements for the SQUEEZE tagset. It should be noted that what we've described above does not account for spanning headers. This and other things may have to be considered to make it more robust. It is left up to the user to add more functionality such as combining more than two columns, having more than one combination column, or allowing for spanning headers.

CONCLUSION

We may never fully understand how the magician pulls the proverbial rabbit out of the hat, but in this paper we have made significant progress. We've gained a more detailed understanding of the relationship between ODS and the PROC. We've seen how the tagset, with a toolbox that contains familiar tools like variables, arrays, conditional and iterative logic, and PUT statements allows us to exploit this relationship in order to directly control the markup without having to manually create the whole file. By choosing examples that focus on customizing output, we have only scratched the surface of what we can do with tagsets. Hopefully it was enough to spark curiosity that leads to further research. With enough practice, users who need their data in a custom XML format may find tagsets the answers to their prayers. Users needing Excel spreadsheets may find the highly parameterized EXCELXP tagset useful. If something is missing from it, add to it. Many users need RTF. Though no usable RTF tagsets exist yet, stay tuned for version 9.2. Want to customize your graphic output? Be sure to research in 9.2 the tagsets that lie underneath the new graphic templates. This paper used HTML for its examples, which meant close examination of HTML markup, but the lessons learned apply to any formats – get to know the events that ODS and the PROCs use and in what order they use them; play detective and use resources such as examination of the markup, mapping tagsets, and text searches of inherited tagsets in order to identify the source of a given piece of markup; and of course, get to know tagset syntax. A mastery of these skills will allow you to dictate the markup without relying on magic.

REFERENCES

Gebhart, Eric "ODS Markup, Tagsets, and Styles! Taming ODS Styles and Tagsets." *Proceedings of the SAS Global Forum Users Group International Conference,* April 2007.

CONTACT INFORMATION

Please feel free to contact me with questions and comments.

Mike Molter
PPD
3900 Paramount Parkway
Morrisville, NC 27560-7200
(919) 462-4199
mike.molter@rtp.ppdi.com