Paper 248-2011

ExcelXP on Steroids: Adding Custom Options To The ExcelXP Tagset

Mike Molter, D-Wise Technologies, Raleigh, NC

ABSTRACT

The multitude of options available with ODS's ExcelXP tagset has allowed users access to dozens of Excel features when creating spreadsheets from SAS®, but not *all* of them. ExcelXP is a SAS-made tool, but because it is a tagset, users have the ability to modify it. In this paper we'll discuss strategies for adding simple functionality to ExcelXP. Users of all levels will not only see the brief, intuitive tagset code used to produce the required XML for these specific examples, but will also realize the power over their output that this and other tagsets give them. Those with more experience with XML and tagset coding will learn a little more about the inner workings of ExcelXP as well as general strategies for adding any functionality to any tagset.

INTRODUCTION

Throughout the years SAS has offered several methods to users for creating output that Excel could read. Using PUT statements in the DATA step, one could write out to a flat file data records in which variable values were separated by delimiters. Telling Excel what those delimiters were would allow it to import it into a spreadsheet. Dynamic Data Exchange (DDE), also through the DATA step, allowed us to write SAS data as well as Excel commands directly to specified cell ranges in specified tabs of existing Excel workbooks. Outside the DATA step, the EXPORT procedure, along with the PROC EXPORT code-generating wizard, was also available. When ODS came along, destinations such as CSV and HTML produced output that Excel could also read. Oftentimes with these technologies, users would have to open the file that SAS created and do some kind of post hoc manipulation of the spreadsheet(s) to finalize what was needed for delivery. What really got people excited though was something called ExcelXP. What was ExcelXP? It wasn't the name of any kind of Microsoft technology like DDE and it wasn't the name of a new SAS PROC. Technically it wasn't the name of a new ODS destination, though in many ways we can think of it that way. Instead, ExcelXP was the name of an ODS *tagset*. Through this new technology, with Excel functionality such as multiple tabs, column filters, several formatting options, and dozens more added throughout the years, SAS was able to get closer and closer to producing delivery-ready spreadsheets that needed no manipulation. But maybe ExcelXP's most valuable gift to us is something that goes beyond any collection of Excel features. Before we go into those details, let's take a quick step back into ODS history.

The introduction of ODS can be thought of as both a blessing and a curse. Finally, users could produce files in formats that were usable by popular programs such as Microsoft Word or web browser programs. These blessings became curses to SAS developers when users began to notice that certain features of these programs weren't accessible through ODS. Through the years SAS became bombarded with requests to add all kinds of functionality to currently existing ODS destinations. In addition to that, more users needed to generate XML markup for which SAS may not have had a destination. Finally, in order to manage all of these requests, SAS came up with a two-part solution that can be summarized in one concise, to-the-point statement – do it yourself! One part of this solution involved allowing users to define templates that instructed ODS how to produce markup. The second part was to get users started on these templates by converting the ODS markup destinations into these templates so that users could have access to them. The name given to these templates was ***tagset***.

So now SAS developers and users can customize the way ODS output is sent to a file, but what does that have to do with Excel? Beginning with Version 2002 of Excel (or the XP version), workbooks could be created with a special form of XML. Since XML is just plain text, every Excel feature simply corresponds to a unique block of text. Since tagsets instruct ODS how to create text, providing Excel support simply meant creating a tagset that wrote the text that Excel reads and uses to give us spreadsheets. Formatting options, multiple tabs, and any other requested features are handled with additions to the tagset. This is the tagset we know as ExcelXP.

With all of the features that the ExcelXP tagset offers us, maybe its greatest gift to us is the fact that it is a tagset; that just as with other tagsets, we can modify it or add to it as we please for our own purposes. Let's face it, as SAS customers, we're accustomed to counting on SAS to build into its functionality the things we need. It's a helpless feeling when we find something was left out. While we wait for the next version of SAS or a hot fix, we're forced to develop temporary workarounds. While ExcelXP offers over 50 Excel features, this falls short of all the features Excel has to offer. We can right-click on a tab in our workbook and change the color of that tab. Since Excel 2002, this corresponds to the addition of simple spreadsheet markup. This functionality is not currently supported in the ExcelXP tagset, but because it is a tagset, we can add it ourselves.

COMING ATTRACTIONS

This paper will demonstrate how to add to ExcelXP support for two specific Excel functions – tab color and tab hiding – but the object of this paper lies in all of the lessons available along the way. Those that are only interested in adding this specific functionality will learn what the associated XML is and where to add it into the tagset. More generally, others may learn something about the spreadsheet XML structure and how the ExcelXP generates it. Though they may not care about these two functions, these users will generate ideas on how they can add other functionality that they need. Even more generally, some may see how to manipulate any SAS tagset or write their own.

The concept sounds simple enough – modifying SAS's template so that the appropriate markup is added to the file, but challenges do exist. For starters, one has to have some knowledge of the markup language they are trying to generate. In general,the amount depends on how much of the tagset you plan to write yourself as opposed to how much you'll inherit from one that already exists. In our case, this translates into knowing something about spreadsheet XML. We also need to know the tagset language – not only the syntax, but the timing of code execution. The remainder of this paper is divided into two major sections. In the first we address these challenges in order to build a complete set of tools. In the second, we develop a general strategy for adding the desired functionality, and then using our new tools, we add it.

SYSTEM REQUIREMENTS

Both Excel and the SAS tagset language have undergone many changes over the years. The tagset language is a relatively new one that continues to see improvements. Additionally, ExcelXP changes seem to be more frequent than SAS version changes. For that reason, updated versions of the tagset are often posted on SAS's website. This paper has been written based on results and tools used with software using the versions mentioned below. Attempts to use other versions may still work, but aren't guaranteed.

- Windows XP
- Excel 2007
- SAS version 9.2
- ExcelXP tagset version 1.94

CHALLENGES

On the surface, if you don't know anything about them already, the challenges might seem daunting. First, we have to learn a new computer language in XML. Even if you know a little about XML, you still have to learn about the particular "brand" of XML that Excel reads. Already this sounds like a lot to ask. If we can manage to get through that, then the second challenge is to learn how to use PROC TEMPLATE to create a tagset. More specifically, it would seem we would have to learn about the ExcelXP architecture if we're going to modify it. Given the reputation of tagsets as well as PROC TEMPLATE, and the complexity of SAS-defined tagsets, this task seems almost insurmountable.

While some XML background knowledge is helpful, all we really need to know is the specific way that the functions we want to add are manifested in the markup. To do this, we don't need any long reference manuals on Spreadsheet XML. All we need to do is to implement the functionality through Excel's familiar spreadsheet interface and observe how the XML is affected. For readers that want some background, in this paper we'll provide some basic facts about XML. We'll also make some observations of how the Spreadsheet XML is structured. Finally, we'll isolate the markup responsible for the functions we want to add, in order to determine what text to generate with a tagset.

Tagsets do come with a reputation of being difficult to understand and to program, and ExcelXP is no exception. As with XML though, we don't need a full comprehensive picture of how tagsets work or how ExcelXP works. We just need to discover what parts of ExcelXP would generate the markup we need. Again, for readers interested in general background, we will go into some detail on how SAS sends data to ODS and how ODS delivers output to a file through tagsets. Though a complete explanation of ExcelXP's structure is beyond the scope of this paper, we will examine in some detail the parts of ExcelXP that we discover need modification. A full discussion of all tools available to the tagset author is also out of scope, but we'll discuss any we find necessary for our purposes. References at the end of the paper will direct you to resources for further information on these topics.

If you're not interested in background, then skip ahead to the section titled EXCELXP.

XML

For the most part, almost everything we learn when we learn a new computer language falls into one of two categories: keywords and syntax. A keyword is simply a collection of characters that has special meaning to a computer language. In SAS we casually refer to IF-THEN statements, the CONTENTS procedure, the MPRINT option. Statements, PROCs, and options are all different instances that make use of a keyword in order to uniquely process a request. Markup languages like HTML are different. Unlike executable languages like SAS, markup languages don't really *do* anything. Instead they are used by other programs. They have keywords that we often refer to as ***tags***, but rather than actively initiate a request for processing, they passively sit there and wait for programs such as web browsers to find them and interpret their meaning. In HTML each tag uniquely corresponds to a display characteristic. For example, any text that a browser finds within a <b> tag is displayed by the browser in bold font. HTML has dozens of different tags and most have one or more allowable ***attributes*** (also specified with a keyword). While learning HTML does require learning some syntax, much of it is learning the meaning of these keywords.

HTML is a markup language that goes hand-in-hand with its corresponding viewing program – a web browser. It is plain text that can be entered into a simple text editor. Without a web browser it would be nothing more than this. Without a web browser, <b> would mean nothing. With a web browser, it contains special keywords that have meaning to the browser. XML, on the other hand, is more generic. In general, XML is a markup language *without* a specific corresponding program, and

therefore, without pre-defined keywords or tags and attributes. Like HTML, XML can also be entered into a text editor, begging the question, without keywords or programs, what good is it? We can think of the answer this way. HTML became HTML when a set of tags and attributes were developed. These keywords became useful when a program was written to use them. XML can be thought of the same way. A "brand" of XML can be developed by defining a set of tags and attributes, along with a program that uses them. Such is the case with the XML that Excel reads. A set of keywords was defined, and Excel is the program written to use these keywords to give us the image and functionality of the spreadsheet that we know.

The second component of a computer language is its syntax. While an XML developer is free to define whatever keywords he wants (subject to certain naming rules similar to those by which we can name datasets) and a program that uses these keywords, all XML documents regardless of the brand must be structured according to a common set of syntax rules. XML syntax revolves mostly around *elements*. Every element begins with a start tag and ends with a corresponding end tag. As mentioned above, tags are uniquely identified by a case-sensitive keyword that we also often use to reference the element. Between these tags an element may contain content or another element. This inner element is sometimes called a *child* element and is said to be nested within its *parent* element. XML requires that nested elements be fully nested, meaning that the start and end tag of a child element be between the start and end tags of its parent element. Every XML file must also contain a *root element*. A root element is one whose start tag is found at the beginning of the file and end tag at the end. All other elements are nested within the root element. Elements may also be further described by one or more attributes, in the form of a name-value pair.

The start tag of any element begins with an open angle bracket followed by the name of the tag. An element with no attributes is then followed by a closed angle bracket.

```
<tag-name>
```

Elements with attributes follow the tag name with the name-value attribute specifications where the value is enclosed in quotation marks, followed by the closed angle bracket.

```
<tag-name name1="value1" name2="value2"…>
```

Elements that contain content follow the closed angle bracket with the content itself, which is then followed by the end tag. End tags contain the name of the tag preceded by a slash, enclosed in angle brackets. They don't contain attribute specifications.

```
<tag-name>content</tag-name>
```

Elements that contain nested elements follow the start tag of the parent element with the start tag of the child element.

```
<parent-tag-name><nested1-tag><nested2-tag>nested 2 content</nested2-tag></nested1-
tag></parent-tag-name>
```

Sometimes an element does not contain content or nested elements. Such an element that may or may not define attributes, is called an *empty* element. As an alternative to having both a start and end tag, empty elements may optionally precede the end of the start tag with a slash.

```
<empty-tag name1="value1"/>
```

Subject to these syntax rules, an XML developer defines not only their keywords, but also which elements contain which attributes (if any), how often a particular element can show up, nesting structures, valid attribute values, and other rules. Such rules are often defined by the developer in a "rulebook" document such as a Document Type Definition (DTD) or an XML schema file. XML files that are intended to be written according to a certain set of definitions often make reference to such a rulebook, and programs written to use the file will sometimes use the rulebook to validate the file.

Markup 1 below illustrates a syntactically correct (also called "well-formed") XML file.

```
Markup 1 – Simple, well-formed XML
<mlb>
<team league="American" division="Central">Tigers</team>
<team league="National" division="West">Giants</team>
<team league="American" division="East">Blue Jays</team>
</mlb>
```

Markup 1 above obeys the syntax rules stated above, and so can legitimately be called XML, regardless of whether anyone has developed rulebooks and applications to use it. HTML can be thought of as one brand of XML where each tag (e.g. `<b>`, `<tr>`, `<th>`, etc) uniquely contributes to the general purpose of this XML brand – web display. While web browsers are

written specifically for this purpose, other XML applications might serve different purposes for different XML brands, such as data extraction and/or storage.

What we've seen here is that learning general XML just means learning a small set of syntax rules. On the other hand, to understand the inner workings of an XML-based application and have the ability to manipulate it for custom purposes requires a knowledge of how the application uses the keywords. This requires a knowledge of the pre-defined element structure – their names, allowable attributes, nesting structures, etc. This is the case in this paper as we try to manipulate through the XML that underlies Excel. So how do we best learn this particular brand of XML?

The beauty of XML-based graphical user-interface (GUI) programs like Excel is that XML is generated as we create and manipulate such files with point-and-click tools. For times like now we can learn about the XML simply by observing through a plain text editor changes that correspond to GUI manipulations. We can start by examining some of the structure of a blank workbook. To do this, simply open Excel, and without doing anything else, choose File, Save As. With the Save as Type drop down box, choose XML Spreadsheet 2003. Give the file a name, save it, and open it with a plain text editor such as Notepad. Figure 1 below shows a portion of the underlying XML, followed by selected observations of the entire file.

Figure 1

```
<Worksheet ss:Name="Sheet1">
 <Table ss:ExpandedColumnCount="1" ss:ExpandedRowCount="1" x:FullColumns="1"
  x:FullRows="1" ss:DefaultRowHeight="15">
 </Table>
 <WorksheetOptions xmlns="urn:schemas-microsoft-com:office:excel">
  <PageSetup>
   <Header x:Margin="0.3"/>
   <Footer x:Margin="0.3"/>
   <PageMargins x:Bottom="0.75" x:Left="0.7" x:Right="0.7" x:Top="0.75"/>
  </PageSetup>
  <Selected/>
  <ProtectObjects>False</ProtectObjects>
  <ProtectScenarios>False</ProtectScenarios>
 </WorksheetOptions>
</Worksheet>
```

- The name of the root element is Workbook (not shown)
- Children of Workbook include DocumentProperties, ExcelWorkbook, Styles, and Worksheet (illustrated in Figure 1)
- Worksheet, whose only attribute (under these conditions) is ss:Name, is an element that appears three times. The values of the ss:Name attribute ("Sheet 1", "Sheet 2", and "Sheet 3") tell us that each instance of Worksheet corresponds to what we see as a tab in the spreadsheet.
- Each Worksheet contains an empty Table element as well as a WorksheetOption element whose descendant elements carry data about page setup and other non-tabular data.

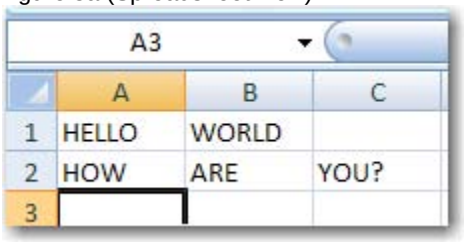
By performing simple Excel functions, we can see how any function is represented in XML. Note in Figure 2 below the change in XML from Figure 1 as a result of simply taking the blank workbook that corresponds to Figure 1 and highlighting cells A1-C1 of Sheet 1.

Figure 2

```
<Worksheet ss:Name="Sheet1">
 <Table ss:ExpandedColumnCount="1" ss:ExpandedRowCount="1" x:FullColumns="1"
  x:FullRows="1" ss:DefaultRowHeight="15">
  <Row ss:AutoFitHeight="0"/>
 </Table>
 <WorksheetOptions xmlns="urn:schemas-microsoft-com:office:excel">
  <PageSetup>
   <Header x:Margin="0.3"/>
   <Footer x:Margin="0.3"/>
   <PageMargins x:Bottom="0.75" x:Left="0.7" x:Right="0.7" x:Top="0.75"/>
  </PageSetup>
  <Unsynced/>
  <Selected/>
  <Panes>
   <Pane>
    <Number>3</Number>
    <RangeSelection>R1C1:R1C3</RangeSelection>
   </Pane>
  </Panes>
  <ProtectObjects>False</ProtectObjects>
  <ProtectScenarios>False</ProtectScenarios>
 </WorksheetOptions>
</Worksheet>
```

To nobody's surprise, when we populate cells of a spreadsheet, the Table element is no longer empty, as seen in Figure 3b below.

Figure 3a (Spreadsheet view)



	A	B	C
1	HELLO	WORLD	
2	HOW	ARE	YOU?
3			

Figure 3b (XML view)

```
<Worksheet ss:Name="Sheet1">
 <Table ss:ExpandedColumnCount="3" ss:ExpandedRowCount="2" x:FullColumns="1"
  x:FullRows="1" ss:DefaultRowHeight="15">
  <Row ss:AutoFitHeight="0">
   <Cell><Data ss:Type="String">HELLO</Data></Cell>
   <Cell><Data ss:Type="String">WORLD</Data></Cell>
  </Row>
  <Row ss:AutoFitHeight="0">
   <Cell><Data ss:Type="String">HOW</Data></Cell>
   <Cell><Data ss:Type="String">ARE</Data></Cell>
   <Cell><Data ss:Type="String">YOU?</Data></Cell>
  </Row>
 </Table>
```

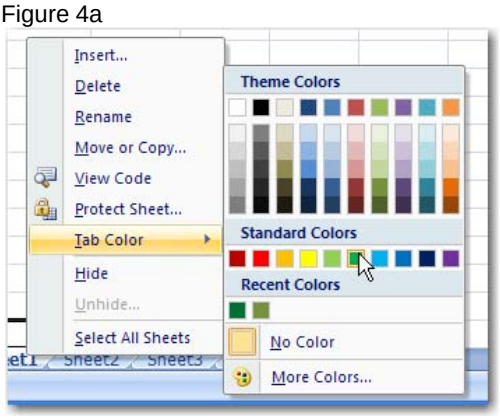
Discovering the XML that controls tab color is just as easy. By right-clicking on the tab of interest and selecting Tab Color, we can color a tab with any color found in the palette. Figure 4 shows the corresponding XML.

Figure 4

```
<WorksheetOptions xmlns="urn:schemas-microsoft-com:office:excel">
 <PageSetup>
  <Header x:Margin="0.3"/>
  <Footer x:Margin="0.3"/>
  <PageMargins x:Bottom="0.75" x:Left="0.7" x:Right="0.7" x:Top="0.75"/>
 </PageSetup>
 <Unsynced/>
 <Print>
  <ValidPrinterInfo/>
  <HorizontalResolution>600</HorizontalResolution>
  <VerticalResolution>600</VerticalResolution>
 </Print>
 <TabColorIndex>21</TabColorIndex>
 <Selected/>
 <Panes>
  <Pane>
   <Number>3</Number>
   <ActiveRow>26</ActiveRow>
   <ActiveCol>1</ActiveCol>
  </Pane>
 </Panes>
 <ProtectObjects>False</ProtectObjects>
 <ProtectScenarios>False</ProtectScenarios>
</WorksheetOptions>
```

While explanation of numeric representations of Excel colors is beyond the scope of this paper, we can see that the TabColorIndex element was added, whose content (or value) is the number that corresponds to the green color chosen in figure 4a. This will be the XML we generate with our modified ExcelXP tagset.

Figure 4a



TAGSETS

Now that we know *what* to generate, we need to know *how* to generate it using SAS. No matter what kind of markup we're dealing with, whether it's Spreadsheet XML, any other kind of XML, HTML, or RTF, the fact that these markups are just plain text might lead us to ask the question, "Why can't we just use a DATA step with PUT statements?" While technically this is possible, many complications make this a long, difficult process. One would have to collect all of the PROC results, all of the necessary system metadata, non-tabular text such as titles and footnotes, system option choices, stylesheet information, and more. All of this would have to be collected in one dataset, ordered in a way that is consistent with the order in which it should be found in the output markup file. Once that is accomplished, PUT statements would have to write out a combination of the data from this dataset and the necessary surrounding markup. On top of this, painstaking care would have to be taken to make sure that all of the tags are closed and that elements are nested properly. What would be ideal is if SAS could do all that for you, leaving you only with the job of choosing options, executing the PROC, and deciding on the kind of markup. That's what ODS does for us.

When SAS first gave us ODS, there was a lot with which to be impressed. On the surface, the differences were obvious. We could produce the same tabular output, but in different file formats that supported style templates. We could customize style templates to give our output any look we wanted. In some cases we could make inline style requests for individual elements to override what was defined in the template. Now we were much closer to producing delivery-ready output, either for word processing programs or for web publishing. Maybe what was more impressive though was what you found behind the scenes. Though our ODS tabular output looked like traditional output with styles attached, this was a product not of SAS but the viewing program (e.g. word processors, web browsers) being used. SAS was no longer producing tables, it was producing text (markup) that these programs displayed as tables. That was the magic of ODS. By taking a simple PROC and surrounding it by two statements that indicated the desired file format, SAS went from producing ASCII text that looked like a table to producing a long text file that would be displayed to look like the ASCII table, but with styles attached. This was good news to users who needed to produce output in a particular format, especially those that didn't know the markup. Even users who did know the markup found it convenient not to have to write it themselves. To those, however, who wanted 100% access to any part of the markup in order to have 100% customization capabilities, it wasn't always as convenient. It was for these users that tagsets were introduced.

We can think of tagsets informally as the secret to the ODS magic tricks. Now users had access to this magic that was once part of SAS's hidden source code. Now we can see how the addition of two simple ODS statements - one before a PROC and the other after - can instruct SAS to replace simple ASCII text with markup text. So what was the secret? Simple, familiar, intuitive, PUT statements. The same PUT statements that we use to write dataset data and literal text to a flat file from a DATA step, along with other familiar DATA step tools such as conditional logic, DO-WHILE, statement blocks, and functions, are also responsible for the same functionality when it comes to writing ODS data to the output file. What gets tricky is access to the ODS data as well as understanding the timing of tagset code execution.

In contrast to the DATA step, the ODS tagset is a *template*, making it a passive piece of code. The DATA step is an active, executable piece of code. We submit the DATA step and statements like SET go out and get data one record at a time. Each record is subject to processing by every executable statement after SET. When we submit a template, rather than executing anything, we are *compiling* it, or making it available to any process that needs it. More specifically, we are compiling a collection of code "pieces", each of which becomes available when needed. When a process makes use of a template, then each part of the process applies the code piece from the specified template with which it is associated. If you've worked with style templates before then you may be familiar with this process. In style templates, each piece is referred to as an *element*, and each element uniquely defines a set of attribute specifications. Every part of displayed output is formatted (through markup) according to the specifications found in the style element with which it is associated.

Tagset code pieces are referred to as *event definitions*. Earlier we mentioned several of the different types of data that must be accounted for to generate a markup file. We then mentioned that SAS gathers all of this data for us automatically and sends it to ODS in the order in which we would expect it to appear in the markup file. When SAS sends this data, rather than

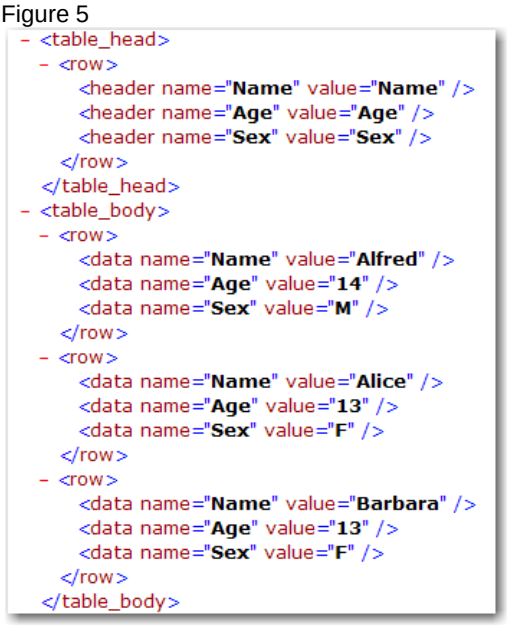
sending it all at once, it sends it in individual units called *events*. The data that it sends is sent through *event variables* and *style variables*. Just as every piece of output must be associated with a set of formatting instructions, every unit of data (event) sent to ODS must be accompanied by a set of instructions for writing it to the output file. A tagset is simply a template with a collection of event definitions, each instructing ODS how to generate markup when SAS sends the corresponding event. Depending on the ODS job, SAS sends a core set of events that can change slightly with circumstances. For example, a BYLINE event will be sent if the PROC has a BY statement. When a given event is sent, ODS executes the code in the corresponding event definition in the tagset. If the tagset doesn't define the event, then nothing happens.

The theory behind tagsets is simple enough but writing a tagset has its challenges. Knowing to define events that SAS sends is one thing but knowing the names of these events is another. The same can be said of the code inside the event definitions – we know that PUT statements can write a combination of literal text and variable values to the file the same way they do in the DATA step, but we don't know the names of the variables we can write. We can execute statements conditionally as in the DATA step, but we don't know the values of the variables upon which conditions can be based. When we write a DATA step to read data we can easily look at that data to know what variables we can reference and what variable values we can use in conditional processing. Tagset code is executed on data such as the results of PROCs. This is data that has traditionally passed from the PROC straight thru to the output file without stopping to give us a chance to look at it. We can't exactly open it with SAS Viewer or Viewtable or execute PROC CONTENTS, FREQ, or PRINT to get to know it. We can, however, execute PROCs using the Markup destination with *mapping tagsets*.

Mapping tagsets, or what we might call "meta-tagsets," produce output that tells us the events that SAS sends, and some of the variables and their values. SAS has provided us with a few of these including SHORT_MAP and EVENT_MAP. Both of these produce XML-formatted output in which tag names are the names of events. Tag attributes from SHORT_MAP show names and values of the VALUE and NAME event variables, while EVENT_MAP shows the values of several additional event variables. Program 1 below uses the SHORT_MAP tagset to illustrate the events sent when a simple PROC PRINT is executed. Figure 5 illustrates a portion of the output, viewed through Internet Explorer.

```
Program 1 – SHORT_MAP tagset output
ods markup tagset=tagsets.short_map file='figure 5.xml' ;
proc print noobs data=sashelp.class (obs=3) ;
var name age sex ;
run;
ods markup close ;
```

Figure 5



From figure 5 we can see that the first time the HEADER event is sent, the value of VALUE (and NAME) is "Name", the second time it's "Age", and the third time it's "Sex". These values as well as their order is a function of the VAR statement. A statement in the HEADER event definition like `PUT VALUE ;` would result in these values being written to the output file.

Before we move on to ExcelXP, there's one more aspect to the *event model*, the name given to the mechanism by which output data is sent to ODS, that we should look at. If you know anything about markup languages, then you probably know that a key part of most markup languages is nesting – the idea that markup that represents an element is often separated by

markup that represents another element. Figure 5 illustrates this concept, but keep in mind that the tags in figure 5 represent event names sent by SAS. Notice too that some of the event names are preceded by a slash (to represent an end tag). How in a tagset definition can we tell ODS to generate a slash sometimes and not others for the same event? In order to support element nesting in generated markup, tagset events can be defined with *start states* and *finish states*. When this is the case, SAS will send an event's start state, followed sometime later by its finish state. Example 5 shows us that after the start state of the TABLE_BODY event is sent, the start and finish states of three instances of the ROW event are sent before TABLE_BODY's finish state is sent. Within each ROW event, three instances of the DATA event are sent. Why? Three DATA events are sent within each ROW event because the PROC only requested three columns of data. Similarly, three ROW events are a function of only three observations being requested (from the OBS= option). Why are DATA events nested within ROW events and ROW events within one TABLE_BODY event? This nesting structure is built into the event model because it mimics typical nesting structure in common markup languages. The event model is characterized in part by the events SAS sends under a given set of circumstances and the order in which they are sent. We now see that an event nesting pattern is also part of the event model's makeup.

EXCELXP

Just like any other kind of program written by someone other than yourself, a tagset written by someone else can be very difficult to follow. The fact that it's a template and that the order in which events are written doesn't have to reflect the order in which data is written to the file makes matters even worse. Adding to the difficulties are *triggered* events. The TRIGGER statement in an event definition executes code in the event named in the statement. For example, we can define in our tagset an event named DIMAGGIO. Though SAS never sends an event of this name, an event that SAS does send might have a TRIGGER DIMAGGIO statement that would execute the code in this event. Trying to follow the flow of a tagset that makes extensive use of the TRIGGER statement is like trying to follow a macro that contains calls to other macros, some or all of which contain calls to other macros, and so on. Many of the SAS-supplied tagsets, especially ExcelXP, are written with all of these traits of complexity. We know that with an understanding of the macro language we can easily understand what individual statements will do, but to understand the overall picture of how a complex macro is to achieve its purposes can require endless time and patience. The same can be said of tagsets.

This isn't meant to discourage you from digging into the trenches of ExcelXP, but the fact that all of the hard work has been done for us and all anyone needs to do is to make minor modifications to a small piece of the puzzle usually makes this exercise unnecessary. Such is the case with our examples. For each, we'll identify the necessary markup to add, find the part of ExcelXP that generates the surrounding markup, discuss any necessary tagset language, and then create our own tagset that inherits ExcelXP and implements the necessary change.

Program 2 below illustrates the production of an Excel workbook with multiple tabs using SAS's ExcelXP.

```
Program 2 – Multiple tab workbook
ods markup tagset=tagsets.excelxp file='age_sheets.xml'
      options(suppress_bylines='yes' sheet_interval='bygroup');

  proc print data=class noobs ;
  by age;
  var name sex height weight ;
  run; quit;

ods markup close;
```

Here we assume that CLASS is the result of sorting SASHELP.CLASS by AGE. The options on the ODS MARKUP statement keep the bylines out of the cells of the spreadsheets, and instead make them the names of each tab. More information on these and other options built into ExcelXP can be found in DelGobbo's paper. The result, as seen through Excel, is seen in Figure 6a while a portion of the associated XML markup is illustrated in Figure 6b below.

Figure 6a

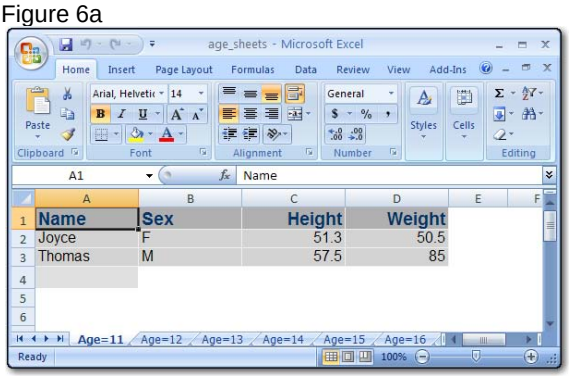


Figure 6b

```
<Worksheet ss:Name="Age=11">
 <Table ss:ExpandedColumnCount="4" ss:ExpandedRowCount="4" x:FullColum
  x:FullRows="1" ss:StyleID="s62" ss:DefaultRowHeight="15">
  <Column ss:StyleID="s62" ss:Width="84" ss:Span="3"/>
  <Row ss:AutoFitHeight="0" ss:Height="18">
   <Cell ss:StyleID="s63"><Data ss:Type="String">Name</Data></Cell>
   <Cell ss:StyleID="s63"><Data ss:Type="String">Sex</Data></Cell>
   <Cell ss:StyleID="s64"><Data ss:Type="String">Height</Data></Cell>
   <Cell ss:StyleID="s64"><Data ss:Type="String">Weight</Data></Cell>
  </Row>
  <Row ss:AutoFitHeight="0">
   <Cell ss:StyleID="s65"><Data ss:Type="String">Joyce</Data></Cell>
   <Cell ss:StyleID="s65"><Data ss:Type="String">F</Data></Cell>
   <Cell ss:StyleID="s66"><Data ss:Type="Number">51.3</Data></Cell>
   <Cell ss:StyleID="s66"><Data ss:Type="Number">50.5</Data></Cell>
  </Row>
  <Row ss:AutoFitHeight="0">
   <Cell ss:StyleID="s65"><Data ss:Type="String">Thomas</Data></Cell>
   <Cell ss:StyleID="s65"><Data ss:Type="String">M</Data></Cell>
   <Cell ss:StyleID="s66"><Data ss:Type="Number">57.5</Data></Cell>
   <Cell ss:StyleID="s66"><Data ss:Type="Number">85</Data></Cell>
  </Row>
  <Row ss:AutoFitHeight="0" ss:Height="18">
   <Cell ss:StyleID="s67"/>
  </Row>
 </Table>
 <WorksheetOptions xmlns="urn:schemas-microsoft-com:office:excel">
  <PageSetup>
   <Header x:Data="The SAS System" ss:StyleID="systemtitle"/>
   <PageMargins x:Left="0.08" x:Right="0.08"/>
  </PageSetup>
  <Print>
   <ValidPrinterInfo/>
   <HorizontalResolution>300</HorizontalResolution>
   <VerticalResolution>300</VerticalResolution>
  </Print>
  <Selected/>
  <ProtectObjects>False</ProtectObjects>
  <ProtectScenarios>False</ProtectScenarios>
 </WorksheetOptions>
</Worksheet>
```

Here again we see all of the markup that corresponds to the “Age=11” worksheet tab, which can be divided into the tabular data and the worksheet options. The worksheet options are further subdivided into *page setup, print*, and *protection* options.

THE TAB COLOR EXAMPLE

We now have the tools we need to add functionality to SAS’s ExcelXP tagset. We’ll illustrate by adding an option that allows a user to choose a color for the tabs. We start by discovering the corresponding XML. We do this by right-clicking a tab, selecting the “tab color” option, and choosing a color for the tab. Figure 7 below illustrates the corresponding markup.

Figure 7

```
<Worksheet ss:Name="Age=11">
 <Table ss:ExpandedColumnCount="4" ss:ExpandedRowCount="4" x:FullColumns=
  x:FullRows="1" ss:StyleID="s62" ss:DefaultRowHeight="15">
  <Column ss:StyleID="s62" ss:Width="84" ss:Span="3"/>
  <Row ss:AutoFitHeight="0" ss:Height="18">
   <Cell ss:StyleID="s63"><Data ss:Type="String">Name</Data></Cell>
   <Cell ss:StyleID="s63"><Data ss:Type="String">Sex</Data></Cell>
   <Cell ss:StyleID="s64"><Data ss:Type="String">Height</Data></Cell>
   <Cell ss:StyleID="s64"><Data ss:Type="String">Weight</Data></Cell>
  </Row>
  <Row ss:AutoFitHeight="0">
   <Cell ss:StyleID="s65"><Data ss:Type="String">Joyce</Data></Cell>
   <Cell ss:StyleID="s65"><Data ss:Type="String">F</Data></Cell>
   <Cell ss:StyleID="s66"><Data ss:Type="Number">51.3</Data></Cell>
   <Cell ss:StyleID="s66"><Data ss:Type="Number">50.5</Data></Cell>
  </Row>
  <Row ss:AutoFitHeight="0">
   <Cell ss:StyleID="s65"><Data ss:Type="String">Thomas</Data></Cell>
   <Cell ss:StyleID="s65"><Data ss:Type="String">M</Data></Cell>
   <Cell ss:StyleID="s66"><Data ss:Type="Number">57.5</Data></Cell>
   <Cell ss:StyleID="s66"><Data ss:Type="Number">85</Data></Cell>
  </Row>
  <Row ss:AutoFitHeight="0" ss:Height="18">
   <Cell ss:StyleID="s67"/>
  </Row>
 </Table>
 <WorksheetOptions xmlns="urn:schemas-microsoft-com:office:excel">
  <PageSetup>
   <Header x:Data="The SAS System" ss:StyleID="systemtitle"/>
   <PageMargins x:Left="0.08" x:Right="0.08"/>
  </PageSetup>
  <Print>
   <ValidPrinterInfo/>
   <HorizontalResolution>300</HorizontalResolution>
   <VerticalResolution>300</VerticalResolution>
  </Print>
  <TabColorIndex>30</TabColorIndex>
  <Selected/>
  <ProtectObjects>False</ProtectObjects>
  <ProtectScenarios>False</ProtectScenarios>
 </WorksheetOptions>
</Worksheet>
```

We note the addition of the <TabColorIndex> element, whose value is the numeric representation of the color we chose (in this case, blue). We also note its location – between the end of the PRINT element and the beginning of the empty SELECTED element. We now locate in the ExcelXP tagset any of the statements responsible for producing any of this

surrounding markup. This will give us a good idea of where to add statements in the ExcelXP tagset for producing our new element.

Figure 8 below shows a portion of the definition of the WORKSHEET event found in the ExcelXP tagset.

Figure 8

```
put "<Print>" NL;
put "<ValidPrinterInfo/>" NL;

trigger do_paperSize;

do /if $scale;
   put "<Scale>";
   put $scale;
   put "</Scale>" NL;
done;

put "<FitWidth>" $pages_fitwidth "</FitWidth>" NL;
put "<FitHeight>" $pages_fitheight "</FitHeight>" NL;
put "<LeftToRight/>" NL /if $left_to_right;
put "<HorizontalResolution>";
put $print_dpi;
put "</HorizontalResolution>" NL;
put "<VerticalResolution>";
put $print_dpi;
put "</VerticalResolution>" NL;
put "<Gridlines/>" NL /if $gridlines;
put "<BlackAndWhite/>" NL /if $blackandwhite;
put "<DraftQuality/>" NL /if $draftquality;
put "<RowColHeadings/>" NL /if $RowColHeadings;
put "</Print>" NL;
```

Because of the circumstances, not all statements in this excerpt had an effect on the markup, but the blue arrows indicate which statements did write markup into figure 7. Adding a PUT statement immediately after `put "</Print>" NL;` should write out the markup for our new element in the place we believe it should be. The easiest way to do this is to write a PROC TEMPLATE that inherits TAGSETS.EXCELXP, copy all of the code from this event, and then simply type in the new PUT statement.

```
Program 3 – adding blue tab color to ExcelXP
proc template ;
define tagset myxl ;
parent=tagsets.excelxp;

define event worksheet ;
/*copy code from this event in ExcelXP here*/

/*insert this after put "</Print>" nl ; */
put "<TabColorIndex>30</TabColorIndex>" nl ;
```

OFFERING PARAMETERS (TAGSET OPTIONS)

Of course this is wonderful if blue (the color corresponding to 30) is the desired tab color. If you've written macros or other applications before, you've had to consider the question of flexibility – among all the parameters, which ones will I allow the user to choose values and which ones will the application choose internally? Each option available to users in ExcelXP (such as SHEET_INTERVAL illustrated in program 2 above) represents a parameter given to the user to decide. In program 3 above the application chose blue for tab color. We'll now see how we can make this another option for the user. In order to do that, we first need to discuss tagset variables.

Tagsets have several different kinds of variables. We've already briefly discussed event and style variables that are populated by the SAS process as it passes PROC, option, and other information to ODS. These are referred to simply by their names as we do in the DATA step. Just as the DATA step allows us to create new variables to supplement those found in the data it reads, tagsets also allow us to create new variables. One kind is called a *memory* variable. The other two, though called variables, are functionally more like arrays. One is called a *list* variable and the other is a *dictionary* variable. All three are referenced with a leading dollar sign ($). Unlike event and style variables whose values represent the event in which they are sent, values of these variables last from the time they're initialized until the end of the ODS job or until they are explicitly deleted.

Character memory variables are normally assigned with the SET statement while numerics are assigned by the EVAL statement. UNSET deletes variables.

```
set $cvar 'This is a character variable' ;
```

```
eval $nvar index($cvar, 's') ;
unset $cvar ;
```

List and dictionary variables (or "array variables", as I call the two of them collectively), are also initialized by the SET statement. Whereas list variables, like our DATA step arrays, are indexed by integers, dictionary variables are indexed by text strings. Unlike DATA step arrays, neither of these has its dimension declared. Elements can simply be added one by one. The dimension or number of elements currently stored in an array variable can be referenced by the name of the variable without an index.

```
set $lvar[1] 'number1' ;
set $lvar[] 'last element';
put 'THE LVAR LIST HAS' $lvar ' elements.';

set $dvar['Raleigh'] 'North Carolina' ;
set $options['TABCOLOR'] '30' ;
```

This last SET statement isn't entirely out of the blue (no pun intended). When options are specified by a user on the ODS statement, they create a dictionary variable called OPTIONS. The name of each option becomes an index to the variable and its corresponding value becomes the value of the dictionary variable for that index. In program 2 above, $options["SUPPRESS_BYLINES"] = 'yes', and $options["SHEET_INTERVAL"] = 'bygroup'. In many ways, tagset options work like macro parameters. Users initiate values when they call on the application, and authors make reference to their names in order to implement the user-chosen values within the application code. Of course macros reference them with preceding ampersands. Tagsets reference them with $options[*OPTION-NAME*]. The only difference is that unlike macro parameters, tagset options don't have to be declared in the tagset definition.

We can use this OPTIONS dictionary to our advantage to allow the user to pick a tab color. In program 4 below we've replaced the new PUT statement from program 3 above that hard-coded a tab color of 30 with code that represents the user's choice.

```
Program 4 – user's choice for tab color
set $tabcolor $options['TABCOLOR'];
set $tabcolor '30' / if ^$tabcolor ;
put '<TabColorIndex>'$tabcolor'</TabColorIndex>' nl ;
```

The first statement in this program creates the memory variable TABCOLOR and assigns to it the user's choice as specified through the TABCOLOR option. The second statement sets a default of 30 to TABCOLOR if TABCOLOR wasn't assigned by the previous statement, which would be the case if the user didn't provide a color. Between these two statements, TABCOLOR will be populated, and its value will be written along with the TabColorIndex tags in the third statement. Program 5 below uses our new tagset to color each of the tabs pink.

```
Program 5 – Pink tabs
ods markup tagset=myxl file='age_sheets_change_pink.xml'
options(suppress_bylines='yes' sheet_interval='bygroup' tabcolor='45');

proc print data=class noobs;
by age;
var name sex height weight ;
run; quit;

ods markup close;
```

THE TAB HIDING EXAMPLE

In our next example we introduce to our tagset the ability to hide one or more tabs. While on the surface the difference between hiding and coloring tabs might not seem significant, there's one necessary aspect of this feature that we left out of the last example. In that example the tagset added markup to color *all* of the tabs the same color. Since nobody wants to hide all of the tabs, in this example we add tagset code that adds tab-hiding markup only for those tabs selected by the user. It's left to the reader to consider how tab selection functionality might be added for tab coloring.

User input for this option will be provided by way of a space-delimited list of numbers, each corresponding to the number of the tab (i.e. 1 for first, 2 for second, etc.) to be hidden. As we consider the tagset code for this, let's imagine for a moment how we might build a macro that requires the same kind of processing. You're forced into some kind of iterative processing, and with each iteration, you have to check if the number of the iteration is found in the user-supplied list or macro parameter. Because this list can be of varying length with any number of elements in it, a %DO-%WHILE or a %DO-%UNTIL construct is necessary to compare each element to the iteration number.

```
%let iterationnumber=1 ;
%let listitem=%scan(&userlist,&iterationnumber,%str( )) ;
%do %while(&listitem^=%str()) ;
       %if &listitem = &iterationnumber %then processing ;
       %let iterationnumber =%eval(&iterationnumber +1) ;
       %let listitem=%scan(&userlist,&iterationnumber,%str( )) ;
%end;
```

With macros we might think of iterative processing as meaning processing that takes place through each iteration of a %DO loop. In our case, as we build an Excel workbook, our tagset builds a WORKSHEET element for each tab created. Put another way, the WORKSHEET event is executed once for each tab. This is our iterative processing. The tagset language also contains DO-WHILE processing. If we can keep track of the tab number being processed, then we can use a similar approach to selecting tabs for processing.

As before, let's first see what kind of markup we need to produce and where. We can do this by opening our Excel workbook, right-clicking on one or more of the tabs, and choosing "Hide". After saving the file, when we examine the underlying markup, we find in the markup that corresponds to hidden tabs just before the PRINT element that we saw in the last example, a VISIBLE element.

```
<Visible>SheetHidden</Visible>
```

Once again, this suggests inserting our tagset code into ExcelXP just prior to the PUT statement that writes the start tag of the PRINT element. Before we do that we need to decide how to capture the tab number. One way we could do this is to define a memory variable whose initial value is 1, and simply increment it by one whenever the event is executed. When we look at the current ExcelXP tagset code though, it appears this is already being done with the following statement.

```
eval $numberOfWorksheets $numberOfWorksheets+1 ;
```

Assuming this memory variable serves the purpose we need, compare the tagset code in program 6 below to the macro code above.

```
       Program 6 – insert tab hiding code into ExcelXP
       do / if $options['TABHIDE'] ;
              unset $thcount ;
              unset $th ;
              eval $thcount 1 ;
              set $th scan($options['TABHIDE'],$thcount,' ') ;
              do / while not missing($th) ;
                     put '<Visible>SheetHidden</Visible>' /
                            if $numberOfWorksheets=inputn($th,'8.') ;
                     eval $thcount $thcount+1;
                     set $th scan($options['TABHIDE'],$thcount,' ') ;
              done ;
       done ;
```

Program 6 emphasizes some of the differences in syntax between tagset code and DATA step code. The entire excerpt is a statement block that ends with the DONE statement instead of END. We also notice that an action to be taken precedes the condition upon which it is based instead of following it. The UNSET statements make sure that these variables' slates are wiped clean before we start to use them. The THCOUNT memory variable serves the same purpose as the macro variable `&iterationnumber` above. The memory variable TH, which corresponds to the macro variable LISTITEM above, extracts an item of the list with the familiar SCAN function. The DO/WHILE loop (note the syntax difference) then checks each item of the numbered list against the tab number and writes the appropriate markup when a match is found. By inserting this into our MYXL tagset, the following produces a workbook whose 3rd and 5th tabs are hidden.

```
       Program 7 – Hide tabs 3 and 5
       ods markup tagset=myxl file='hide3and5.xml'
       options(suppress_bylines='yes' sheet_interval='bygroup' tabhide='3 5');

       proc print data=class noobs;
       by age;
       var name sex height weight ;
       run; quit;
       ods markup close;
```

As is the case with macros, when we write tagsets and offer user flexibility through parameters, we want those parameters to be as easy to use as possible. In the example above, we ask the user to know exactly where in the order in which tabs are displayed, a particular tab is found. From a development point of view, we could offer this because we could capture this information in the tagset. Users might argue however that it's not always easy to know, for example, when your PROC produces several tabs, where exactly a specific tab is. Another alternative might be to ask the user for the name, or part of the name, of the tab. To implement this, the tagset author has to find where this is captured in the orginal ExcelXP tagset. Once accomplished, the changes to Program 6 are minimal.

We found the creation of the variable `$numberofworksheets` in the WORKSHEET event. If we expand our search beyond this event, then we find the variable `$worksheetname` in the WORKSHEET_LABEL event. If the name of the variable itself doesn't give away its purpose, then the following code found in this event gives us a good idea that this is the memory variable that holds the name of the worksheet.

```
set $worksheetname $worksheetname $byval_name "=" $last_byval ;
```

For our last example, let's ask the user when using `sheet_interval = "bygroup"` to indicate tabs to hide by providing not the entire name of the tab, but the by-value represented by the tab. With AGE as a by-group, a user would hide tabs representing 12- and 13-year olds with the following:

```
tabhide = "12;13"
```

Here we've switched from using the space to using the semicolon as a delimiter. When allowing users to provide multiple variable values in a parameter, we want to make sure that the delimiter isn't something we expect to show up as a character in variable values. A semicolon might be more rare than a space in variable values, but an even better solution might be to let the user specify through a parameter a delimiter. We'll stick with the semicolon here, but the user is challenged to improve upon it in whatever way they see fit. Below is the new assignment of the `$th` variable in Program 6 with the new delimiter.

```
set $th scan($options['TABHIDE'],$thcount,';') ;
```

The main statement in Program 6 that needs modifying is the PUT statement whose execution depends on the worksheet number and its equality with the user choice. Prior to that, since this condition is based only on part of the name of the worksheet, we should create a variable that holds that part. We know that when the sheets are defined by by-groups, the name of the tab takes on the form *by-variable-name=by-variable-value*. In capturing the value of the by-variable, we could use the SCAN function to extract the text that shows up after the equal sign. In most cases that should work, but if the value of the by-variable itself contains an equal sign, then we won't capture the entire value that way. The following code addresses that possibility.

```
eval $whereisequal index($worksheetname,"=")+1 ;
set $worksheetvalue substr($worksheetname,$whereisequal) ;
```

All that's left now is to change the condition of the PUT statement.

```
put '<Visible>SheetHidden</Visible>' / if $worksheetvalue=$th ;
```

CONCLUSION

Excel is a powerful program with many tools available to present data in ways that helps its consumers to make decisions. If that data is coming from SAS, then the more of those tools we can send along with the data, the fewer manual manipulations that we or the consumers have to perform on the back end. ExcelXP has brought us closer to that objective than we've ever been with Excel. Luckily, through ODS tagsets, SAS has shared the code used to give us this functionality, allowing us to add whatever additional tools our data consumers need. More generally, with tagsets, SAS has given us the opportunity to intercept any data flow originating from SAS en route to the external file. Depending on the final format, we may have something to start with, courtesy of SAS. Such was the case in this paper, and we were able to get away with using simple searches to discover where additions needed to made. We also had the Excel program to tell us what markup to add. On the other end of the spectrum though, other situations might require a full knowledge of the markup keywords and the rules that dictate where they belong. To write a tagset from scratch, a complete understanding of ODS's event model may be required. Luckily, programming logic tools used to manipulate data from the event model are similar to those in the DATA step. Although we can't learn about the data we're manipulating in traditional ways, mapping tagsets give us the same opportunity with output data.

REFERENCES

DelGobbo, Vincent. 2009. "More Tips and Tricks for Creating Multi-Sheet Microsoft Excel Workbooks the Easy Way with SAS ®." *Proceedings of the Third Annual SAS Global Forum Users Group International Conference,* Washington D.C.,

Molter, Mike. 2008. "A Tiptoe Through the Tagset Field." *Proceedings of the Second Annual SAS Global Forum Users Group International Conference,* San Antonio, TX.

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

RECOMMENDED READING

Molter, Mike. Forthcoming in spring, 2011. "SAS ODS Tagsets: Mastering the Markup Destination"

CONTACT INFORMATION

Please feel free to contact me with any questions or comments in any of the following ways.

Mike Molter
919-463-9453 (home)
molter.mike@gmail.com