Paper 014-30

Tagset Spelunking and Cartography: Debugging and Exploring Tagsets with Battery-Powered Headlamps

Eric A. Gebhart, SAS Institute Inc., Cary, NC

ABSTRACT

Tagsets are perceived as a dark maze of twisty passages. But there are many ways to look inside and map the passages and their contents. The key to understanding and using the tagsets is knowing how to look inside. There are several simple methods that will shine the light on everything from the simplest of tagset changes to the creation of very complex tagsets.

INTRODUCTION

From the surface, looking at Output Delivery System (ODS) MARKUP and tagsets can be like looking into a deep, dark cave that has a sign outside that says 'Here Be Dragons!'. Well, even dragons that live in deep, dark caves need light—and the myth says that they like jewels and other nice things too. Who knows? Maybe there are no dragons at all; maybe there are just jewels and a well-equipped tool shed for excavation.

I discovered most of these deep, dark caves—and I know what is in them, mostly. Tagsets are the basis for the most powerful and flexible ODS destinations ever. The examples presented in this paper show that tagsets are not dark, dank caves with monsters and dragons. The caves are actually well-lit and jewel-encrusted with plenty of tools for mining. The batteries in your headlamp are just low.

This presentation will give you a basic understanding of how tagsets work, how to explore and map them, and how to debug them. When you understand all this about tagsets, you'll be able to use them in new and unimagined ways. So, throw away your headlamps, you don't need them here. Tagsets shed all the light that we need to enter and explore the caves.

ENTERING THE CAVE – UNDERSTANDING TAGSETS

There is a lot of documentation about tagset syntax. There are also several explanations of how tagsets work (see The ODS MARKUP Resources Web site at http://support.sas.com/rnd/base/topics/odsmarkup). The most important thing to understand about tagsets is that you have to use them if you want to understand them. It also helps to know that tagsets use an event-driven model, which means that a tagset is a collection of event definitions. When a tagset receives an event request, it does what its event definition tells it to do. A tagset doesn't read like a procedural language, from top to bottom. Although a tagset event definition can have procedural elements such as conditions and loops, that's the extent of its relationship to more traditional languages.

The following example code defines a tagset that has **data** and **header** events. The **data** event corresponds to each value in an observation, unless the value is in an ID column. The **header** event corresponds to each value that is in an ID column or is part of the headers for a table. VALUE is the most important event variable. It holds key information for any given event. A value can be almost anything such as a data value, a header, a title, or a byline.

```
proc template;

    define tagset tagsets.example1;

       define event header;
           put "Header: " value nl;
       end;

       define event data;
           put "Data: " value nl;
       end;

       define event row;
           start:
               put "==================" nl;
           finish:
```

```
                put "==================" nl;
        end;
    end;
run;

options obs=2;

ods tagsets.example1 file="example1.txt";
proc print data=sashelp.class;

run;

ods _all_ close;
```

The output from this job is predictably simple. PROC PRINT produces a report that organizes the data by a series of row, data, and header calls for each observation.

```
==================
Header: Obs
Header: Name
Header: Sex
Header: Age
Header: Height
Header: Weight
==================
==================
Header:  1
Data: Alfred
Data: M
Data: 14
Data: 69.0
Data: 112.5
==================
==================
Header:  2
Data: Alice
Data: F
Data: 13
Data: 56.5
Data:  84.0
==================
```

The preceding example is rather simplistic, but it's a first step. Let's go deeper into the cave. We can refine the tagset to be more versatile by using another event variable called EVENT_NAME. Here is the refined tagset.

```
proc template;

define tagset tagsets.example1;

        define event basic;
                put event_name ": " value nl;
        end;

        define event header;
                trigger basic
        end;

        define event data;
                trigger basic
        end;

        define event row;
                start:
                put "==================" nl;
                finish:
```

```
            put "==================" nl;
        end;
    end;
    run;
```

This tagset adds the new event **basic**, which is an arbitrary name that was created for our own purposes. Events can be called by other events so that the code can be centralized.

The output from this tagset is exactly the same as the output from the previous tagset. Now it's time to explore these events.

GATHERING YOUR EQUIPMENT

In this discussion, exploring means looking at the information that is available in these events. We get this information by using event variables. There are approximately 170 pre-defined event variables and approximately 110 pre-defined style variables to choose from in SAS 9.1 and later. It is also possible to create your own variables, which can be strings, numbers, or streams. Lists and dictionaries of numbers and strings can also be created. All these variables can be used in a PUTVARS statement.

THE PUTVARS STATEMENT

The PUTVARS statement is an implicit loop through a given namespace, list, or dictionary. There are five namespaces:

- The **event** namespace contains all the metadata about the event.
- The **style** namespace corresponds directly to the ODS style that is currently in use.
- The **memory** namespace holds all the variables that are created within the tagset by using a SET or EVAL statement.
- The **stream** namespace holds all the stream variables that are created in the tagset by using an OPEN statement.
- The **dynamic** namespace holds all dynamic variables and is primarily used with SAS/GRAPH software.

Generally, **stream** and **dynamic** namespaces are not useful. Streams can be large, so printing them is not something that you would usually do. Dynamic variables are somewhat unpredictable. The **event**, **style**, and **memory** namespaces are the most useful. Of these namespaces, the **event** namespace contains the most useful information. Although style variables are somewhat suppressed because of style sheets, adding them doesn't usually hurt readability. Memory variables are most useful in situations in which a complex tagset is not working as expected. Often, all it takes to find the problem is to add a PUTVARS statement to the **memory** namespace at a key point in the tagset.

THE XENON EVENT

Replacing the basic event with the new **xenon** event will enable us to get more output. The **xenon** event looks like this.

```
define event xenon;
    put event_name nl;
    putvars event "   " _name_ ": " _value_ nl;
    putvars style "   " _name_ ": " _value_ nl;
    put "------------------" nl;
end;
```

Because the output from this job is too large to show in its entirety, only a small part of it is shown.

```
==================

header: Obs
  name: Obs
  dname: Obs
  label: Obs
  anchor: IDX
  colstart: 1
  row: 1
  colwidth: 3
  event_name: header
  encoding: iso-8859-1
  operator: saseag
  date: 2004-12-01
  sasversion: 9.2
  saslongversion: 9.02.01B0D11182004
  time: 18:07:17
  section: head
  state: start
  col_id: 1
  trigger_name: basic
  value: Obs
  proc_count: 1
  total_proc_count: 1
  page_count: 1
  total_page_count: 1
  data_row: 0
  dest_file: body
  bodyname: example1.txt
  tagset: TAGSETS.EXAMPLE2
  style: Default
  data_viewer: Report
  style_element: Header
  HTMLCLASS: Header
  NAME: Obs
  FRAMEBORDER: auto
  CONTENTSCROLLBAR: auto
  BODYSCROLLBAR: auto
==================
```

THE HALOGEN EVENT

As you can see, the preceding output contains a lot of information, but much of the information never changes such as DATE, TIME, SASVERSION, BODYNAME, and OPERATOR. There are a few variables that are important: NAME, LABEL, VALUE, COLSTART, ROW, and STYLE_ELEMENT. Instead of using a PUTVARS statement, we can use the **halogen** event that prints a subset of these variables, or we can use the **xenon** event to print all the variables. Using the **xenon** event can be overwhelming if you use it all the time. Here's the tagset that contains the two new events. It's important to put a marker at the beginning and end of these events so that the output doesn't run together. It can be very confusing without markers.

```
proc template;

    define tagset tagsets.example4;

        define event xenon;
           put "********************" nl;
           putvars event "   "  _name_ ": " _value_ nl;
           putvars style "   "  _name_ ": " _value_ nl;
           putvars mem "   "  _name_ ": " _value_ nl;
           put "********************" nl;
       end;

       define event halogen;
```

```
            put "value: " value nl;
            put "label: " label nl;
            put "name: "  name nl;
            put "htmlclass: " htmlclass nl;
            put "section: " section nl;
            put "anchor: " anchor nl;
        end;

        define event basic;
            put event_name ": " nl;
            trigger halogen;
            put "------------------" nl;
        end;

        define event header;
            trigger basic;
        end;

        define event data;
            trigger basic;
        end;

        define event row;
            start:
                put "==================" nl;
            finish:
                put "==================" nl;
        end;
    end;
run;
```

The output from this tagset is better, but still fairly large. Here are the first two headers from the output for this example.

```
==================
header:
value: Obs
label: Obs
name: Obs
htmlclass: Header
section: head
anchor: IDX
------------------
header:
value: Name
label: Name
name: Name
htmlclass: Header
section: head
anchor: IDX
==================
```

THE DEFAULT EVENT

The readability of the preceding output can be substantially improved if it is changed to XML. XML is not always easy to read, but it is more compact and predictable. Using EVENT_NAME works well for creating the tag names. At the same time, it might also be useful to add indention so that you can see the nesting of each event. Adding indention requires adding a finish section to the data and header events. Otherwise, the indention will go beyond the page margins. Here is the new tagset.

```
proc template;

    define tagset tagsets.example5;

        indent = 2;

        define event xenon;
           putvars event " " _name_ '="' _value_ '"';
           putvars style " " _name_ '="' _value_ '"';
           putvars mem   " " _name_ '="' _value_ '"';
       end;

       define event halogen;
           putq " value=" value;
           putq " label=" label;
           putq " name="  name;
           putq " htmlclass=" htmlclass;
           putq " section=" section;
           putq " anchor=" anchor;
       end;

       define event basic;
           start:
               put "<" event_name;
               trigger halogen;
               put ">" nl;
               ndent;
           finish:
               xdent;
               put "</" event_name ">" nl;
       end;

       define event header;
           start:
               trigger basic;
           finish:
               trigger basic;
       end;

       define event data;
           start:
               trigger basic;
           finish:
               trigger basic;
       end;

       define event row;
           start:
               trigger basic;
           finish:
               trigger basic;
       end;
    end;
run;
```

The new output is much easier to look at. The indention enables you to better understand the event structure. Of course, any browser will do that for you, now that the output is XML.

```
<row htmlclass="Table" section="head" anchor="IDX">
  <header value="Obs" label="Obs" name="Obs" htmlclass="Header" section="head" anchor="IDX">
  </header>
  <header value="Name" label="Name" name="Name" htmlclass="Header" section="head" anchor="IDX">
  </header>
  <header value="Sex" label="Sex" name="Sex" htmlclass="Header" section="head" anchor="IDX">
  </header>
  <header value="Age" label="Age" name="Age" htmlclass="Header" section="head" anchor="IDX">
```

```
  </header>
  <header value="Height" label="Height" name="Height" htmlclass="Header" section="head" anchor="IDX">
  </header>
  <header value="Weight" label="Weight" name="Weight" htmlclass="Header" section="head" anchor="IDX">
  </header>
</row>
<row htmlclass="Table" section="body" anchor="IDX">
  <header value=" 1" name="Obs" htmlclass="RowHeader" section="body" anchor="IDX">
  </header>
  <data value="Alfred" label="Name" name="Name" htmlclass="Data" section="body" anchor="IDX">
  </data>
  <data value="M" label="Sex" name="Sex" htmlclass="Data" section="body" anchor="IDX">
  </data>
  <data value="14" label="Age" name="Age" htmlclass="Data" section="body" anchor="IDX">
  </data>
  <data value="69.0" label="Height" name="Height" htmlclass="Data" section="body" anchor="IDX">
  </data>
  <data value="112.5" label="Weight" name="Weight" htmlclass="Data" section="body" anchor="IDX">
  </data>
</row>
<row htmlclass="Table" section="body" anchor="IDX">
  <header value=" 2" name="Obs" htmlclass="RowHeader" section="body" anchor="IDX">
  </header>
  <data value="Alice" label="Name" name="Name" htmlclass="Data" section="body" anchor="IDX">
  </data>
  <data value="F" label="Sex" name="Sex" htmlclass="Data" section="body" anchor="IDX">
  </data>
  <data value="13" label="Age" name="Age" htmlclass="Data" section="body" anchor="IDX">
  </data>
  <data value="56.5" label="Height" name="Height" htmlclass="Data" section="body" anchor="IDX">
  </data>
  <data value=" 84.0" label="Weight" name="Weight" htmlclass="Data" section="body" anchor="IDX">
  </data>
</row>
```

THE EVENT_MAP AND SHORT_MAP TAGSETS

If you have ever used any of the mapping tagsets that are shipped with SAS, the preceding and subsequent output probably looks familiar. For examining events and finding data, the mapping tagsets are the way to go. Creating a custom tagset for mapping specific events is also a common thing to do. Using a custom tagset (as we did in the last example) gives you a narrow focus that can make it much easier to see the relationship of specific events and the key values in those events.

The next task is to expand what the tagset shows. Inheritance works nicely for this. The new tagset inherits from the old tagset and adds to it. The first example that follows adds a default event so that all events will show up. The default event is what makes the mapping tagsets work. The EVENT_MAP, SHORT_MAP, TEXT_MAP, and PYX tagsets are the simplest of all the tagsets. The following tagset simply sets the **basic** event as the default event.

```
proc template;
    define tagset tagsets.example6;
        parent = tagsets.example5;
        default_event = "basic";
    end;
run;
```

The beginning of the resulting output looks like this. The data and header events disappear into the other output because they also use the default event. The tagset that's used above is roughly equivalent to using the mapping tagsets EVENT_MAP and SHORT_MAP.

```
<initialize anchor="IDX">
</initialize>
<doc htmlclass="Body" anchor="IDX">
  <doc_head htmlclass="Body" anchor="IDX">
```

```
  <doc_meta htmlclass="Body" anchor="IDX">
  </doc_meta>
  <auth_oper htmlclass="Body" anchor="IDX">
  </auth_oper>
  <doc_title htmlclass="Body" anchor="IDX">
  </doc_title>
  <stylesheet_link anchor="IDX">
  </stylesheet_link>
  <javascript htmlclass="Body" anchor="IDX">
    <startup_function htmlclass="StartUpFunction" anchor="IDX">
    </startup_function>
    <shutdown_function htmlclass="ShutDownFunction" anchor="IDX">
    </shutdown_function>
  </javascript>
 </doc_head>
 <doc_body htmlclass="Body" anchor="IDX">
  <proc name="Print" anchor="IDX">
  .
  .
  .
```

When I'm about to start on a tagset project, I always add an ODS mapping destination to the sample program. Usually, I start with SHORT_MAP and switch to EVENT_MAP if I need more information.

```
ods tagsets.short_map file="map.xml";
ods tagsets.event_map file="map2.xml";
```

I will most likely need to look at the map for a variable, an event, or both. The maps are very convenient for comparing the new or old tagset output. The maps show if there is another, better-placed event, or if some unknown but useful metadata is available. Maps also show inconsistencies that are generated by various procedures in the events or the metadata.

The default event can be used as a tool for finding information. The information that we are trying to reveal is easier to see if we narrow its scope, and change its output format so that it can be easily seen in the midst of the other output. Using square brackets ([ ]) instead of angle brackets (<>) helps. This is particularly true when creating HTML or XML output that can blend in with other output. Uppercasing the event name really makes the default events stand out. The following tagset uses the event variable EMPTY to further decrease the amount of output. The variable EMPTY indicates when an event is, in XML terms, empty. In other words, it will never contain other events. Doc_meta, Auth_oper, and doc_title are examples of this type of event. Using the **bread_crumbs** event as the default will cause unseen events to fill in the gaps between known events. It also creates an XML-like empty tag with no closing tag. This gives us slightly cleaner output. It's also useful information that can be seen at a glance.

```
proc template;
    define tagset tagsets.example7;
        parent = tagsets.example5;
        default_event = "bread_crumbs";

        define event bread_crumbs;
            start:
                put "[" upcase(event_name);
                trigger halogen;
                put "/" / if empty;
                put "]" nl;
                break / if empty;
                ndent;
            finish:
                break / if empty;
                xdent;
                put "[/" upcase(event_name) "]" nl;
        end;

    end;
run;
```

An excerpt of the output follows. It's easy to see the defined events among all the events that have been exposed by the default **bread_crumbs** event.

```
.
.
.
 [/TABLE_HEADERS]
[TABLE_HEAD anchor="IDX"]
 <row htmlclass="Table" section="head" anchor="IDX">
  <header value="Obs" label="Obs" name="Obs" htmlclass="Header" section="head" anchor="IDX">
  </header>
  <header value="Name" label="Name" name="Name" htmlclass="Header" section="head" anchor="IDX">
  </header>
  <header value="Sex" label="Sex" name="Sex" htmlclass="Header" section="head" anchor="IDX">
  </header>
  <header value="Age" label="Age" name="Age" htmlclass="Header" section="head" anchor="IDX">
  </header>
  <header value="Height" label="Height" name="Height" htmlclass="Header" section="head" anchor="IDX">
  </header>
  <header value="Weight" label="Weight" name="Weight" htmlclass="Header" section="head" anchor="IDX">
  </header>
 </row>
[/TABLE_HEAD]
[TABLE_BODY anchor="IDX"]
 <row htmlclass="Table" section="body" anchor="IDX">
  <header value=" 1" name="Obs" htmlclass="RowHeader" section="body" anchor="IDX">
  </header>
  <data value="Alfred" label="Name" name="Name" htmlclass="Data" section="body" anchor="IDX">
  </data>
  .
  .
  .
```

It's sometimes very useful to expose more information for specific events. Using the **xenon** event works well in this case. We can also use inheritance to keep this simple and non-invasive. Setting the default event to a non-existent event definition minimizes the output by showing only the events that are defined, which makes it easier to see what we want to see. The following tagset contains the new event **basic_plus**. It's the basic event with `trigger xenon` substituted for `trigger halogen`. You can also add `trigger xenon` to an event that can be placed conveniently if only it had the data that you're looking for. Using the **xenon** event helps you to find the data that you're looking for or to identify missing data. The following example shows what to do if you think that using the **proc** event might be useful. This example defines the **proc** event to trigger the new, very verbose **basic_plus** event.

```
proc template;
    define tagset tagsets.example8;
        parent = tagsets.example7;
        default_event = "INVALID";

        define event basic_plus;
            start:
                put "<" upcase(event_name);
                trigger xenon;
                put "/" / if empty;
                put ">" nl;
                break / if empty;
                ndent;
            finish:
                break / if empty;
                xdent;
                put "<" upcase(event_name) "/>" nl;
        end;

        define event proc;
            start:
                trigger basic_plus;
            finish:
```

```
                trigger basic_plus;
            end;

        end;
    run;
```

The output is now much simpler without all the noise of the uninteresting events. The **proc** event is prominently displayed. One of the more useful variables for this event is PROC_NAME. Even more useful is it's timing. The timing and relationship of events is just as important as the information that they might contain.

<PROC name="Print" anchor="IDX" event_name="proc" encoding="iso-8859-1" operator="saseag" date="2004-12-01" sasversion="9.2"

saslongversion="9.02.01B0D11182004" time="22:07:11" state="start" trigger_name="xenon" proc_name="Print" dest_file="body" bodyname="example8.xml"

tagset="TAGSETS.EXAMPLE8" style="Default" javadate="2004-12-01" javatime="22:07:11-05:00" style_element="default" HTMLCLASS="Body" FRAMEBORDER="auto"

CONTENTSCROLLBAR="auto" BODYSCROLLBAR="auto">
 <row htmlclass="Table" section="head" anchor="IDX">
  <header value="Obs" label="Obs" name="Obs" htmlclass="Header" section="head" anchor="IDX">
  </header>
  <header value="Name" label="Name" name="Name" htmlclass="Header" section="head" anchor="IDX">
  </header>
  <header value="Sex" label="Sex" name="Sex" htmlclass="Header" section="head" anchor="IDX">
  </header>
  <header value="Age" label="Age" name="Age" htmlclass="Header" section="head" anchor="IDX">
  </header>
  <header value="Height" label="Height" name="Height" htmlclass="Header" section="head" anchor="IDX">
  </header>
  <header value="Weight" label="Weight" name="Weight" htmlclass="Header" section="head" anchor="IDX">
  </header>
 </row>
 <row htmlclass="Table" section="body" anchor="IDX">
  <header value=" 1" name="Obs" htmlclass="RowHeader" section="body" anchor="IDX">
  </header>
  <data value="Alfred" label="Name" name="Name" htmlclass="Data" section="body" anchor="IDX">
  </data>

At this point in the process, the **proc** event probably will be defined appropriately or discarded as not being useful. If it is really not useful, then defining an empty **proc** event will cause it to disappear from the output when the default event is set back to **basic** for further examination of the available events.

Another technique is to use the **basic** event as the default. However, instead of letting the default event print everything, set the default so that it only prints events in which some variable has a specific value. This technique is particularly useful for debugging.

The following tagset simplifies the output that's generated by the default event by revealing only the structure of the events. It's completely uncluttered by the data. This tagset is very helpful when working on a new tagset that needs a table of contents or a high degree of structure such as some XML formats. The **initialize** event sets up some new variables that will be used to determine if the default event should print them. This limits the output to include only the interesting parts.

```
proc template;
    define tagset tagsets.example8a;
        default_event = "basic";

        indent=2;

        define event initialize;
            set $title_events "system_title system_footer proc_title byline
```

```
pagebreak";
            set $structure_events "doc proc proc_branch branch bygroup leaf
output table";
            set $event_list $structure_events $title_events;
        end;

        define event basic;
            start:
                break /if ^contains($event_list, event_name);
                put "<" event_name;
                trigger halogen;
                put "/" / if empty;
                put ">" nl;
                break / if empty;
                ndent;
            finish:
                break /if ^contains($event_list, event_name);
                break / if empty;
                xdent;
                put "<" event_name "/>" nl;
        end;

        define event halogen;
            putq " value=" value;
            putq " label=" label;
            putq " name="  name;
            putq " htmlclass=" htmlclass;
            putq " anchor=" anchor;
        end;
    end;
run;
```

The resulting file shows only the structure of the ODS output. None of the extraneous data is shown.

```
<doc htmlclass="Body" anchor="IDX">
  <proc name="Print" anchor="IDX">
    <system_title value="The SAS System" htmlclass="SystemTitle" anchor="IDX">
    <system_title/>
    <proc_branch value="Print" label="The Print Procedure" name="Print" htmlclass="ContentProcName"
anchor="IDX">
      <leaf value="Data Set SASHELP.CLASS" label="Data Set SASHELP.CLASS" name="Print"
htmlclass="ContentItem" anchor="IDX">
       <output label="Data Set SASHELP.CLASS" name="Print" anchor="IDX">
         <table htmlclass="Table" anchor="IDX">
         <table/>
       <output/>
     <leaf/>
   <proc_branch/>
  <proc/>
<doc/>
```

By using the following tagset, I found a resolution to the inconsistent problem of missing values in the excelXP tagset. The problem was easy to see because the output only contained missing values. I added the **proc** and **table** events to provide context to the **data** events. If I didn't add the **proc** and **table** events, the output would be a long list of missing values with no clue as to where they came from.

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

        indent=2;

        define event proc;
            start:
                put "Proc: " name ", " label nl;
                ndent;
            finish:
```

```
            xdent;
        end;

        define event table;
            start:
                put "Table: " name ", " label nl;
                ndent;
            finish:
                xdent;
        end;

        define event data;
              break /if ^missing;
              put "========================";
              putvars event _name_ ": " _value_ nl;
        end;

    end;
run;
```

THE PUTLOG STATEMENT

There is one last fundamental technique for exploring tagsets, that is, using the PUTLOG statement. The PUTLOG statement writes output directly to the SAS log. It works the same as the PUT statement in the regular tagset. The PUTLOG statement is very helpful for looking at specific values within an event; it's also good for the basic 'I am Here!' or 'This is this.' type of debugging messages.

Tagset destinations have an option that's named OPTIONS. OPTIONS are any list of key/value pairs. It's up to the tagset to use them as it desires. Using the PUTLOG statement is a convenient way to watch these values to see if they are set the way that you expect them to be set, especially when the code that uses them isn't working. The following tagset shows how to use the PUTLOG statement to see which options are available. Essentially, this example is the equivalent of using the PUTVARS statement, but the output goes to the log instead of the output file.

```
proc template;
    define tagset tagsets.example12;

    define event initialize;
        iterate $options;
        do /while _name_;
            putlog _name_ ": " _value_;
            next $options;
            done;
        end;
    end;
run;

ods tagsets.example12 file="example12.empty" options(doc="help" events="table
row data");
```

The log from the preceding ODS statement contains the following lines.

```
NOTE: Writing TAGSETS.EXAMPLE12 Body file: example12.empty
DOC: help
EVENTS: table row data
```

REVIEWING YOUR TOOLKIT

You now have the fundamental tools that are needed for exploring the internal world of tagsets. Here are the key tools that we identified.

- The PUTVARS statement
- A xenon event (for convenience)
- The default event for finding the 'between' events
- The EVENT_MAP and SHORT_MAP tagsets
- The PUTLOG statement

USING YOUR TOOLS CORRECTLY

Now that you have the fundamental tools that you need to explore tagsets, knowing how to use them is just as important as knowing what they are. Here are some basic tasks for solving any problem with tagsets.

1. Define the desired result.
2. Identify the key events.
3. Explore the key events.
4. Define or re-define events, as necessary.

Repeat these tasks, as necessary. Writing or modifying a tagset is an iterative process. Even the simplest modifications might take a few passes at finding and exploring before all the required components are found.

DEFINING THE DESIRED RESULT

Defining the desired result is the most obvious task. A reasonable understanding of the desired output is required before any other tasks are performed. A result can be something as simple as modifying an HTML tag in one of the current HTML tagsets, or it could be an entirely new XML format that you create. Either way, a good understanding of tagsets is required.

IDENTIFYING THE KEY EVENTS

Adding one of the mapping tagsets to your job should be the first step toward identifying the events that need defining or modifying. If this is an entirely new tagset, then the maps might be your best resource. If this is a tagset modification, then simply looking through the original tagset might yield the correct events for modification.

If you wanted to modify an HTML **table** event, the easiest course of action is to search through the HTML tagset for 'table'. The first event that's found will most likely be the event that you want to modify. Modification can also be a useful tool if your entirely new, built-from-scratch, XML tagset has a construct similar to the way that HTML constructs its tables.

Using inheritance can also be a handy tool. If you are modifying an existing tagset, then the new tagset should be inheriting from that tagset. However, if this is an entirely new tagset, then inheritance isn't really needed. Yet, it's sometimes convenient to inherit from the SHORT_MAP tagset. It's the same concept that was used previously with the **bread_crumb** event. This is especially useful in the early stages because it shows the relationship of the newly defined events to events that aren't being used but could be useful. The inheritance can be added and removed as the new tagset takes shape.

EXPLORING THE KEY EVENTS

Exploring is the process of identifying the useful variables that are provided by an event. Most events don't use that many variables. One of the problems is that some events might not provide enough metadata on their own. When that happens, it becomes necessary to look for other events that might provide the extra data that's needed.

The first step when exploring is to look for the identified event in the mapping tagset's output. The map will show you more variables and events than most tagsets use. If it's obvious that the identified event satisfies all needs, then code the event the way that you ultimately want it. For modifications, it's best to copy-and-paste the original and work from there. If information is missing, then more exploring is necessary.

If some metadata is missing, then create a new event such as the **xenon** event (see the "Xenon Event" section presented earlier in this paper). Run the job with the new tagset and look at the output. If you find everything you need, then it's time to code the event. If you don't find everything, then it's time to identify and explore some other events. Look at the map to see if there are events around or preceding the target event that might provide the context or metadata that's missing from the event that's being worked on.

DEFINING OR RE-DEFINING EVENTS

This task is the goal of all the other tasks. After an event has been identified and it's variables have been explored, it's time to define what is really desired of this event. If more exploration is needed, it might be desirable to code a partial solution and return to it later. This step can be as simple as modifying a copy of the same event from another tagset.

IDENTIFYING AND EXPLORING – A SIMPLE EXAMPLE

Last year, someone posted a question to the SAS-l mailing list asking if there is a way to list graph images and their names as generated by the GPLOT procedure with a BY group. Numerous replies suggested using various DATA step programs. None of the programs were very long, but some were a bit cryptic, and all required two passes over the data. It's better to use a tagset, which can list graph images and their names with one pass. It's not all that difficult. I had an idea that the **image** event would probably be part of the solution, but I wasn't sure. The following test case is all that's needed to figure it out.

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

goptions reset=all noprompt device=gif;

proc gplot data=sashelp.class;
by name;
plot weight*height;
run;

ods _all_ close;
```

The resulting map.xml has everything needed to identify the necessary events to create a list of images based on the BY values. The first step is to search `map.xml` for .gif. This is a guaranteed match because the images are gif's. The only match is the **image** event.

```
<image event_name="image" trigger_name="attr_out" output_name="Gplot"
output_label="Plot of Weight * Height" index="IDX" just="c" url="gplot.gif "/>
```

This is great, but it doesn't have a name; at least not a name that's useful. A new tagset that looks like this might help. However. that's unlikely because NAME, LABEL, and VALUE aren't showing up in the EVENT_MAP tagset.

```
proc template;
    define tagset tagsets.show_image;
       define event image;
             putvars event _name_ ": " _value_ nl;
       end;
    end;
run;
```

After using the new tagset, there's still no help with the name. Time to look back at the map again. The value of NAME in the first observation is `Alfred`. A search for `Alfred` shows up only in one place—the **branch** event.

```
<branch event_name="branch" trigger_name="attr_out" class="ByContentFolder"
value="Name=Alfred" name="ByGroup1" label="Name=Alfred" index="IDX" just="c"

url="map.xml#IDX" hreftarget="body">
```

The **branch** event contains the **image** event. That means that the tagset will know the name before it knows the image file's name. That's all we need to know. Here is a simple tagset that prints out a list of data names and the associated file name. It even does a little extra and strips the 'name=' from the beginning of the name.

```
proc template;
    define tagset tagsets.gifs2;

         define event branch;
            set $name scan(label, 2, '=');
         end;

         define event image;
            put url "  " $name nl;
         end;
    end;
run;
```

Using this tagset with the previous job creates the following output.

```
gplot.gif  Alfred
gplot1.gif  Alice
gplot2.gif  Barbara
```

The essential skills for working with tagsets involve using tagsets to reveal what is useful about any given event. There is the high-level view provided by the SHORT_MAP tagset, and a very verbose, high-level view is given by the EVENT_MAP tagset.

It's also useful to create a very specific tagset that only displays a small number of interesting events. This is still a mapping tagset, but it's view is very specific. Another mapping technique is to add a mapping tagset as a parent or to add a **default** event. Either method will result in the exposure of currently undefined events.

Another method of focusing on events is to suppress or enable event printing based on the values of interest. After events are found or if a problem is suspected, further illumination can be provided by PUTVARS or by triggering an event that prints the variables in question either to the output file or to the log.

All these things are so common that there's a special mapping tagset that does all this. It's called SuperMap, and it's available at http://support.sas.com/rnd/base/topics/odsmarkup/.

SuperMap takes two different event lists as arguments. Both lists can have different verbosities, from the very illuminating **xenon** to the somewhat less brilliant **halogen** down to a softwhite, and finally no illumination but the event name itself. The tagset also takes regular expressions for VALUE and LABEL. If provided, only events that match the regular expressions will be printed. The SuperMap tagset has practically eliminated the need for the various mapping tagsets.

When it comes to debugging, the skills are the same, but illuminating problems is much simpler. Triggering a xenon-like event or just adding a simple PUTLOG statement at the right time (or times) is often the only tactic needed. The most important key is to label the messages so they are discernable from one another.

CONCLUSION

The examples presented in this paper demonstrate how tagsets work and ways to explore, map, and debug them. These skills are the everyday skills that you need to create and modify ODS destinations that are based on tagsets.

The cave is well lit. No headlamps are required. Tagsets are self-illuminating if you know how to use them. I challenge you to explore these examples and create new ones. The best way to learn tagsets is to use them. Tagsets will enable you to solve tasks, which you previously thought were impossible or very complex, more simply than ever before. Tagsets will illuminate new paths and caverns that lead to new ideas.

RECOMMENDED READING

SAS Institute Inc. 2003. *SAS 9.1 Output Delivery System: User's Guide.* .Cary, NC: SAS Institute Inc.

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact

Eric Gebhart
SAS Institute Inc.
SAS Campus Drive
Cary, NC 27513
Phone: (919) 677-8000

eric.gebhart@SAS.com