ODS Markup:
Tagsets by Example

Eric Gebhart
SAS Institute

September 22, 2003

Preface

The SAS Output Delivery System's (ODS) markup destination appears to be much like all the other destination types that ODS is capable of. In reality it is quite different.

The other ODS destinations have a fixed output type. The RTF destination can only create RTF output. The printer destination can create postscript, PCL, and PDF. But ODS Markup can create any number of output types, all of which can be created or modified by anyone.

ODS Markup is able to do this because it is really a framework which uses a tagset to define what it should print to it's files. The other destinations, including the old HTML destination were hard coded and compiled as a part of the SAS executable.

A tagset can be described as a group of events or functions which use a programming syntax similar to other programming languages such as perl, python, shell, and datastep.

We can create and use as many different tagsets as we like. There are a number of tagsets that come standard as a part of SAS.

Because there are so many different tagsets, ODS Markup is rather chameleon-like. ODS HTML is really ODS Markup. So is the CSV destination, and CHTML, excelXP, LaTeX, Troff, etc. The only difference between these different ODS destinations is the tagset that is in use. They are all really ODS Markup, we just don't call it that.

When we use ODS Markup we are using a tagset. The tagset determines the type of output. Therefore, a tagset defines an output destination! This realization has far reaching implications. It means that we can change the html that ODS HTML generates. It means we can create a new HTML destination with your corporate style and headings. We can create a new XML destination to allow data interchange with a business partner or client. We can even create a simple flat file format that can simplify processing by other programs. The output can be modified and adapted in ways that could only be imagined until now. The possiblities are endless.

It stands to reason, that if we want to use ODS Markup to our best advantage, what we really want to do is use tagsets. That is what this book is going to explore. What are these tagsets and how do we use them? You will find that tagsets can simplify otherwise complex problems in a way that allows reuse and flexibility that would not be possible with out them.

Contents

20 ODS Output for Website Integration 201
20.1 The Problem . . . 201
20.2 Alternate Behavior for existing options . . . 202
20.2.1 Explore . . . 202
20.3 Reading an external file . . . 202
20.4 The Solution . . . 206
20.4.1 Initialization Timing . . . 206
20.5 Summary . . . 210

21 Datastep Conversions 213
21.1 Special Bylines . . . 213
21.1.1 The DataStep Code . . . 213
21.1.2 Breaking it down . . . 215
21.1.3 The Style . . . 217
21.1.4 Counting Observations . . . 217
21.1.5 various problems . . . 221
21.1.6 Modifying the Byline . . . 223
21.1.7 A more flexible solution . . . 225
21.2 Summary . . . 229

List of Tables

Part I

Using ODS Markup and Tagsets

Chapter 1

Introduction

ODS markup and tagsets are the best way to create markup output from SAS. This book is divided into sections that will enable you to get the most out of ODS Markup and Tagsets. The first section shows how ODS Markup and Tagsets are used. Using an ODS destination that is defined by a tagset is no harder than using any other destination. The most popular ODS Markup destinations are covered in this first section.

The next sections dig deeper into how Tagsets work and what you can do to modify them or create your own. This may seem daunting at first, but frequently it is quite easy to get Tagsets to do exactly what you want, saving lots of time and effort further down a project's path.

Chapter 2

The basics

This chapter will cover the basics of using ODS Markup family of destinations. There are a lot of ODS destinations in this family, and they are ready to use. By the end of this chapter you will have some idea of the destinations available, and how to use them.

ODS Markup is just another ODS destination that works very much like every other ODS destination. It was originally derived from the original ODS HTML destination, so if you are familiar with ODS HTML, then you are already very familiar with ODS Markup.

The way ODS Markup differs is that we will almost never use ODS Markup as a destination name. ODS Markup should be thought of as a family of destinations. These destinations are defined using a type of template called Tagsets. Each Tagset definition is a new ODS destination.

The good news is that there are already a lot of Tagsets defined, so all we need to do is use them. Since Tagsets define destinations within the ODS Markup family the only thing we really need to know is their names. Once we know the tagset's name we know the destination name, and we can easily construct an ODS statement that will use that ODS destination.

2.1 It's all in the Name

A tagset is a SAS program that uses Proc Template. There are a lot of advantages to this. It means that SAS can update a tagset, make a fix, and even email that tagset to anyone that wants it. Updates to the Tagsets that shipped with SAS are available for download on the Support.sas.com website. Frequently there are also additional tagsets that were not shipped with SAS. Another benefit, is that if you want to learn to program Tagsets, you can create your own, or modify the Tagsets provided by SAS.

For now, the most important thing to know, is what are the Tagsets available, and what are they named. All we really need is their names. But where do we look? When a Tagset is run in SAS, Proc Template compiles the Tagset and stores it When a Tagset is run in SAS, Proc Template the Tagset it into a binary format which is then stored on disk in an itemstore. Typically this file will be in sasuser. But can be placed anywhere using the 'ODS path' statement. If administrative priviledges are available the template can be put in SASHELP where it will be available to all users on the system. Once a template is compiled there is no need to compile it again unless it's itemstore is deleted. All Tagsets that ship with SAS are in SASHELP.

Thankfully, Proc Template handles all of this for us, and will look for Tagsets in all the right places. All we need is 3 lines of SAS code, and we will have all the names for all the Tagsets.

2.1.1 The tagset directory

Templates are kept in directories within the itemstore. ODS will automatically look for tagsets in the tagsets directory. The following example shows how to get a list of tagsets, and the output it generates.

Listing 2.1: Simple ODS Markup Statement

```
proc template;
    list tagsets;

                                The SAS System
1
                                                    13:00 Monday, August 7, 2006

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

                Obs    Path                              Type
                 1     Tagsets                           Dir
                 2     Tagsets.Cascading_stylesheet      Tagset
                 3     Tagsets.Chtml                     Tagset
                 4     Tagsets.Colorlatex                Tagset
                 5     Tagsets.Config_debug              Tagset
                 6     Tagsets.Csv                       Tagset
                 7     Tagsets.Csvall                    Tagset
                 8     Tagsets.Csvbyline                 Tagset
                 9     Tagsets.Default                   Tagset
                10     Tagsets.Docbook                   Tagset
                11     Tagsets.Event_map                 Tagset
                12     Tagsets.ExcelXP                   Tagset
                13     Tagsets.Graph_rtf                 Tagset
                14     Tagsets.Html4                     Tagset
                15     Tagsets.Htmlcss                   Tagset
                16     Tagsets.Htmlpanel                 Tagset
                17     Tagsets.Imode                     Tagset
                18     Tagsets.Latex                     Tagset
                19     Tagsets.MSOffice2k                Tagset
                20     Tagsets.Mlatex                    Tagset
                21     Tagsets.Mvshtml                   Tagset
                22     Tagsets.Namedhtml                 Tagset
                23     Tagsets.Odsapp                    Tagset
                24     Tagsets.Odsgraph                  Tagset
                25     Tagsets.Odsstyle                  Tagset
                26     Tagsets.Odsxrpcs                  Tagset
                27     Tagsets.OpenOffice_rtf            Tagset
                28     Tagsets.Phtml                     Tagset
                29     Tagsets.Pyx                       Tagset
                30     Tagsets.Rtf                       Tagset
                31     Tagsets.Rtf_sample                Tagset
                32     Tagsets.SASReport11               Tagset
```

```
                33      Tagsets.SASReport12                 Tagset
                34      Tagsets.SASReport13                 Tagset
                35      Tagsets.SASReport14                 Tagset
                36      Tagsets.SASReport15                 Tagset
                37      Tagsets.SASReport_html              Tagset
                38      Tagsets.SASReport_html1             Tagset
                39      Tagsets.Sasreport_html11            Tagset
                40      Tagsets.Short_map                   Tagset
                41      Tagsets.Simplelatex                 Tagset
                42      Tagsets.Style_display               Tagset
                43      Tagsets.Style_popup                 Tagset
                44      Tagsets.Supermap                    Tagset
                45      Tagsets.Tablesonlylatex             Tagset
                46      Tagsets.Text_map                    Tagset
                47      Tagsets.Tpl_style_list              Tagset
                48      Tagsets.Tpl_style_map               Tagset
                49      Tagsets.Troff                       Tagset
                50      Tagsets.Wml                         Tagset
                51      Tagsets.Wmlolist                    Tagset
                52      Tagsets.Xbrl                        Tagset
                53      Tagsets.Xhtml                       Tagset
        run;

NOTE: PROCEDURE TEMPLATE used (Total process time):
      real time           1:40.15
      cpu time            0.28 seconds
```

2.2 Using a tagset

That is a lot of Tagsets! For now, lets concentrate on the obvious ones. To use a tagset, all that is needed is the proper ods statement. ODS Markup is not all that different from ODS HTML, in fact, ODS Markup can do everything ODS HTML can, plus a little more. The simplest form of the ODS Markup statement is shown in Figure 2.2 on page 7.

Listing 2.2: Simple ODS Markup Statement

```
ods markup file='test.xml';

....

ods markup close;
```

The result of those statements will be the creation of the 'test.xml' output file. The output of that file will have a format as defined by the default tagset, which is named, tagsets.default.

But there are other ways to specify an ODS statement that accomplish the same thing. Figure 2.3 on page 7. shows ods statements which are equivalent with one exception.

Listing 2.3: ODS Markup Statement Permutations

```
\index{ods markup statement!permutations}

    ods markup file='test.xml';

    ods markup tagset=default file='test2.xml';

    ods tagsets.default file='test3.xml';
```

The exception is the third statement. That statement is different because the destination name is no longer markup. The destination name is tagsets.default. Because it has a unique name it can run simultaneously with ods markup. Just like ODS RTF can run simultaneously with ODS HTML. The first two statements cannot be used simultaneously because they have the same destination name. The second ODS Markup statement would automatically close the first one. This happens because ODS destination names must be unique. There is a way around that, using ODS' ID syntax, but that belongs in another book. Beside's that syntax is mostly uneccessary if we use the tagset name as the destination name, and the resulting code is more readable and concise.

As of SAS 9.1, ODS HTML is also a tagset. Which means ODS HTML is really ODS Markup. Consider the statements in figure 2.4 on page 8.

Listing 2.4: ODS Markup Statement Permutations

```
    ods html file='test1.html';

    ods html4 file='test2.html';

    ods tagsets.html4 file='test3.html';

    ods markup tagset=html4 file='test4.html'
```

Each one of those statements is roughly equivalent. All of them use the html4 tagset. So the markup each of them creates will be the same. But each ODS statement creates a unique output destination. They can all run simultaneously without conflict. They can all have their own select or exclude statements and be opened or closed at different times.

There is one thing that makes html special. Normally a tagset cannot be referenced by it's simple name unless the tagset option is used. But there are a few special tagsets that can be referenced as if they are a simple ODS destination. Table 2.1 on page 9 shows the list of shortcut names and their descriptions.

While these are special, the custom tagsets that can be written are no less special. These custom tagsets cannot be referenced by their simple names but they will be output destinations just as these are. Because Markup is a single destination, the prefered way to use tagsets is to use their two part name. There is a trend away from using these shortcut names, so most of the newer tagsets like ExcelXP and RTF can only be referenced by their full name. See Figure 2.5 on page 8 shows the list

Listing 2.5: Multiple Active Markup Destinations

```
ods tagsets.ExcelXP file='test.xml';
ods tagsets.RTF file='test.rtf';

proc print data=sashelp.class;
run;
```

Table 2.1: Shortcut Destination names

HTML	Alias for HTML4
HTML4	4.0 compliant HTML with concessions for browser compatibility.
PHTML	Plain HTML. Simple stylesheet, simple HTML.
HTMLCSS	4.0 compliant HTML with fewer concessions.
CSV	Comma separated values. Tables only.
CSVAll	CSV with titles, bylines, notes, etc.
CSVByline	CSV with bylines only.
WML	Wireless Markup Language. - No longer in vogue.
CHTML	Compact HTML. Very simple HTML with no styles.
LaTeX	LaTeX output with style and color support.
Troff	troff output which is very simple.
MSOffice2K	HTML for importing into Word and Excel.
Imode	Compact HTML that has no table tags. Used extensively in Japan.
DocBook	XML format for documents.
SasReport	XML format used internally by other SAS products.

```
ods tagsets.ExcelXP close;
ods tagsets.RTF close;
```

2.3 Summary

Using ODS Markup does not have to be complicated or difficult, all we really need is the name of the Tagset, and we can use them just like any other destination. Finding their names is very simple Proc Template code, and adding an ods statement that uses them is just as easy. It won't hurt anything to try out a destination just to see what it will create. Go ahead, play, try them out. The next chapters will go into more detail for many of the more popular and useful Tagsets that are available.

The most confusing thing about the ods markup statement is that it is almost never used by it's name. Aside from that it is almost the same as any other ods destination, especially the ODS HTML destination in previous releases of SAS.

Chapter 3

Using The CSV Destinations.

The CSV Tagsets are some of the most useful tagsets around, the format has been used for decades by various applications and everyone knows what to expect from a CSV file. This chapter will explain how to use the 3 different CSV tagsets provided by SAS.

Comma separated values are one of the simplest formats that are is understood by many different applications. The disadvantage is that there is no formatting, fonts, or colors. Still CSV files work very well for many applications. A CSV file can be connected as live data to an Excel Template. This can be a convenient way to publish Excel spreadsheets to many people.

There are three different CSV Tagsets shipped with SAS. The simplest is the CSV Tagset, next comes CSVByline, and last is CSVAll. The CSV Tagset only creates output from tabular data, there are no titles, footnotes, bylines, Notes, there is nothing but tables. It didn't take long to find out that there were people who wanted titles and footnotes and everything else, that's when CSVAll came into existence. It also became apparent that just having tables and bylines was a nice thing to have, so CSVBylines was born. All of these tagsets use the same base Tagset to create the CSV data, so there is no other difference in behavior besides the verbosity of the output.

Both CSV and CSVAll have aliases so that they can be referenced by a simple name, rather than the full tagset name. CSVByline has no alias, so it must be used with it's full name. Here are examples of each of these ODS statements.

```
ODS CSV file='test.csv';
ODS CSVAll file='testall.csv';
ODS tagsets.CSVByline file='testby.csv';
```

Using these statements around a simple Proc Print will show how these destinations vary.

```
ODS CSV file='test.csv';
ODS CSVAll file='testall.csv';
ODS tagsets.CSVByline file='testby.csv';

options obs=6;

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

Proc print data=sorted;
```

```
      by sex;
run;

ODS _all_ close;
```

None of the output from this example is very complicated, but each has it's differences. The CSV Tagset ignores the title and the bylines. But it does create the two tables as a result of the by.

```
"Obs","Name","Age","Height","Weight"
"1","Alice",13,56.5,84.0
"2","Barbara",13,65.3,98.0
"3","Carol",14,62.8,102.5

"Obs","Name","Age","Height","Weight"
"4","Alfred",14,69.0,112.5
"5","Henry",14,63.5,102.5
"6","James",12,57.3,83.0
```

The CSVByline Tagset adds the bylines, but nothing else.

```
"Sex=F"
"Obs","Name","Age","Height","Weight"
"1","Alice",13,56.5,84.0
"2","Barbara",13,65.3,98.0
"3","Carol",14,62.8,102.5

"Sex=M"
"Obs","Name","Age","Height","Weight"
"4","Alfred",14,69.0,112.5
"5","Henry",14,63.5,102.5
"6","James",12,57.3,83.0
```

Finally, the CSVAll tagset adds in the title. If there were footnotes, or procedure titles, or any of the four flavors of notes; Note, Warning, Error, or Fatal, those would be included as well.

```
The SAS System

"Sex=F"
"Obs","Name","Age","Height","Weight"
"1","Alice",13,56.5,84.0
"2","Barbara",13,65.3,98.0
"3","Carol",14,62.8,102.5

"Sex=M"
"Obs","Name","Age","Height","Weight"
"4","Alfred",14,69.0,112.5
"5","Henry",14,63.5,102.5
"6","James",12,57.3,83.0
```

In the beginning it seemed that this was everything that any CSV tagset would ever need to do. But even the simplest of outputs is not always that simple. Tagsets needed to be able to adapt to what was needed of them. The best way to do that is to have the ability to give the tagset some variable to help it make decisions on how to behave. Up until SAS 9.1.3 the only way to do that was with macro variables. Macro variables

still work, but now there is a better way. With SAS 9.1.3 the ODS Markup statement was given a new option, the options(...) option. Options are a nearly arbitrary set of key value pairs that the tagset will recieve. It is completely up to the Tagset to process those values and use them.

Surprisingly the CSV Tagset has quite a number of options. In fact everything that the CSVAll and CSVByline tagsets do can be done by the base CSV Tagset. In SAS 9.1.3 the only thing CSVALL and CSVByline Tagsets do is set up some new defaults for the CSV options available.

With the proliferation of Tagset options it didn't take long to figure out we needed another option, Help! All Tagsets that have options have an option called doc. The Doc option is all you need to find out just what a Tagset can do. Asking for help may be the first thing you want to do when using a new tagset. If there are options, Help will tell you about them. Help isn't in all of the Tagsets, but the number is growing and it never hurts to ask. Here is how we ask for help.

```
ODS CSV file='test.csv' options(doc='help');
```

The output from help will go to the log and for CSV, it looks like this.

```
================================================================================
The CSV Tagset Help Text.

This Tagset/Destination creates output in comma separated value format.

Numbers, Currency and percentages are correctly detected and show as numeric values.
Dollar signs, commas and percentages are stripped from numeric values by default.

================================================================================

These are the options supported by this tagset.

Sample usage:

ods csv options(doc='Quick');

ods csv options(currency_as_number='yes' percentage_as_number='yes' delimiter=';');

Doc:  No default value.
     Help: Displays introductory text and options.
     Quick: Displays available options.

Delimiter:   Default Value ','
     Sets the delimiter for the values.  Comma is the default.  Semi-colon is
     a popular setting for european sites.

currency_as_number:   Default Value 'No'
     If 'Yes' currency values will not be quoted.
     The currency values are stripped of punctuation and currency symbols
     so they can be used as a number.

percentage\_as\_number:   Default Value 'No'
     If 'Yes' percentage values will not be quoted.
     The percentages are stripped of punctuation and the percent sign
     so they can be used as a number.
```

```
Currency\_symbol:   Default Value '\$'
     Used for detection of currency formats and for
     removing those symbols so excel will like them.
     Will be deprecated in a future release when it is
     no longer needed.

Decimal\_separator:   Default Value '.'
     The character used for the decimal point.
     Will be deprecated in a future release when it is no longer needed.

Prepend\_Equals:   Default Value 'no'
     Put an equal sign in front of quoted number values.
     This only works in conjunction with quote_by_type.

quote\_by\_type:   Default Value 'no'
     Put values based on the type, not based on what the regex match for number.

Table\_Headers:   Default Value 'yes'
     If no, skip the header section of all tables.

Thousands\_separator:   Default Value ','
     The character used for indicating thousands in numeric values.
     Used for removing those symbols from numerics so excel will like them.
     Will be deprecated in a future release when it is no longer needed.

Quoted\_columns:   Default Value ''
     A list of column numbers that indicate which values should be quoted
     ie. Quoted\_columns="123"

Empty\_Missing:   Default Value 'no'
     If yes, missing values will not show in any way other than positionally.

Bylines:   Default Value: No
     If yes bylines will be printed

Titles:   Default Value: No
     If yes titles and footnotes will be printed

Notes:   Default Value: No
     If yes Note, Warning, Error, and Fatal notes will be printed

Proc\_Titles:   Default Value: No
     If yes titles generated by the procedures will be printed
```

The help you get will depend upon which version of the tagset you have. There are the expected options that reflect the behaviors we have seen from the CSVAll and CSVByline Tagsets, but there is much more. One of the most used options is Delimiter, this makes it easy to change the field delimiter to a ;, a | or even tabs. Another thing that became obvious was that there needed to be a way to detect numbers in any one of the various European formats. That's where decimal_separator, thousands_separator and currency_symbol

come in. Not everyone uses a '.' for decimals, or ',' for thousands.

There are a few less obvious options in there. Table_headers turns the table_headers on and off. Empty_missing makes missing values go completely away, with just it's field delimiters showing where it would have been.

Currency_as_Number, and Percentage_as_Number, are there because not everyone wants those values as strings, So if you want them as numbers, these options will do that for you, they will also strip out any symbols that would prevent Excel from seeing those values as numbers.

Another option that is just for excel, is prepend_equals, this option will put an '=' in front of quoted numbers so that Excel will read the value correctly, even if it has leading zeros. So if you have leading zero's this is the option you want.

The last two have to do with quoting. Currently, the CSV tagset tries it's best to figure out what is a string and what is a number. It doesn't always do exactly what we want. Part of the problem is that some procedures don't tell the Tagset what is what. Proc Print is one of those procedures that does say which value is a string and which is a number. The quote_by_type option tells the tagset to stop guessing and do what Proc Print or any other proc says. The exception to this, until SAS 9.2, is Proc Report, and Proc Tabulate. Those are the two Procs that don't give the tagset any clues.

To further extend the ability to control quoting, There is a Quoted_columns option that will allow for columns to be specified by number. So if all else fails and you still can't get your values quoted where you want, this option should do the trick.

By default the csv tagset only displays tables. Turning on bylines so that it looks like the output from the CSVByline Tagset is just a matter of setting the bylines option to yes. Changing the Delimiter to a ';' is just as easy. Of course we could have used the CSVByline Tagset with just the delimiter option to do the same thing.

```
ODS CSV file='test.csv' options(delimiter=';' bylines='yes');
ODS tagsets.CSVByline file='test.csv' options(delimiter=';');

options obs=6;

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

Proc print data=sorted;
     by sex;
run;

ODS _all_ close;
```

The output from both of these ODS statements looks identical.

```
"Sex=F"
"Obs";"Name";"Age";"Height";"Weight"
"1";"Alice";13;56.5;84.0
"2";"Barbara";13;65.3;98.0
"3";"Carol";14;62.8;102.5

"Sex=M"
"Obs";"Name";"Age";"Height";"Weight"
"4";"Alfred";14;69.0;112.5
"5";"Henry";14;63.5;102.5
"6";"James";12;57.3;83.0
```

Knowing how the CSVByline and CSVAll Tagsets work can be quite handy, Especially if you find that you always want the same basic set of options. It is much easier to create your own Tagset than it is to always type in the same options over and over. Here is the entire code for the CSVByline Tagset. Before you look, realize that most of this is copied and pasted from the CSV Tagset.

```
define tagset tagsets.csvbyline;
    parent=tagsets.csv;

     define event initialize;
         set \$options['BYLINES'] 'yes';
         trigger set_options;
         trigger documentation;
         trigger compile_regexp;
     end;
 end;
```

The only addition to the copied and pasted code is this line.

```
         set \$options['BYLINES'] 'yes';
```

We can take advantage of this knowledge and create our own CSV Tagset that has it's own set of default option values. If we want the same behavior from the new Tagset as our previous example a new name is probably in order. After all it's no longer comma separted values, it's semi-colon separated values. Our new Tagset would look like this. Be aware that option names are always capitalized inside the tagset.

```
proc template;
   define tagset tagsets.ssvbyline;
       parent=tagsets.csv;

        define event initialize;
            set \$options['BYLINES'] 'yes';
            set \$options['DELIMITER'] ';';
            trigger set_options;
            trigger documentation;
            trigger compile_regexp;
        end;
    end;
run;
```

All we have to do is run that Proc Template, and our new tagset is ready to use. Congratulations! You've just written your first Tagset! If someone else wants output that has bylines and semi-colons for field separators, just tell them to do this!

```
ODS tagsets.ssvbyline file='test.csv' options(doc='help');
```

The output looks exactly like the output above. It is, after all, generated the same way.

3.1 Summary

The CSV destinations are very easy to use and although the output is very simple, there are many options that can be used to control how that output is created. ODS Markup options provide a very versatile way to change the behavior of any Tagset destination. Finding out what those options are, and how to use them is

as easy as asking for help. Add options(doc='help') to any ODS Markup/tagsets statement and find out just what that Tagset can do for you. Creating a new Tagset that sets the options just the way you want is almost as easy is copy and paste. That's what tagsets are all about, creating output just the way you want it, in a reusable and easy way.

Chapter 4

Using The HTML Tagsets

HTML is certainly one the most popular output typs available. Not only is easily used to report information on the web, but it is also a commonly understood import format for many applications. ODS Markup has several HTML tagsets to choose from when it comes to creating HTML output. Each of these Tagsets has some advantage in one way or another depending upon how it will be used. This chapter will explain the different HTML tagsets and the features they support.

4.1 The Different HTML's

ODS HTML is one of the aliased destination names that really means use the tagsets.html4 Tagset. Someday it may well mean use the Tagsets.html5 Tagset. But that is in the future.

There are several other aliases, for the various HTML tagsets. PHTML, HTMLCSS, CHTML and CHTML_imode are the other shortcut names for HTML tagsets. These aliases map directly to their tagset names, the aliases just keep us from having to put 'tagsets.' in front of them.

Only one example is needed in order to see the major differences between the different HTML tagsets available. There are differences in style, the way titles and footnotes are processed, and in the case of the MSOffice2k tagset, there is a difference in how the tables are rendered. All that is needed to see this differences is a couple of title statements and some simple Proc Tabulate output.

This example has three titles, the default title, and two more titles that use the 3 part title syntax. It is the use of style and 3 part titles that will show major differences between these various HTML destinations.

```
ods html file="html.html";
ods xhtml file="xhtml.html";
ods htmlcss file="htmlcss.html";
ods phtml file="phtml.html";
ods tagsets.MSOffice2K file="msoffice2k.html";
ods chtml file="chtml.html";
ods imode file="imode.html";


  title3 height=15pt color=orange
         justify=right '15 pt first right'
         justify=left 'first left';
```

```
  title4 height=10pt color=red 'no justify red'
         justify=right 'Second right'
         justify=left 'Second left';

proc sort data=sashelp.class out=work.class;
    by age  sex;
run;

options obs=2;

PROC TABULATE DATA=class;
   VAR Height Weight;
   CLASS Sex Age;
   TABLE Age*Mean ALL*Sex*Mean,
         Weight;
   by age;

RUN;

ods _all_ close;
```

4.1.1 HTML4 And XHTML

HTML4 and HTML are really the same thing and XHTML is nearly the same but not quite. The only thing the XHTML tagset does is add XML compliant tags to the HTML. If you don't know what that is, don't worry about it. For all practical purposes they are nearly identical.

The HTML destination does not have to be HTML4, it could be set to be HTML3. There is a registry setting that will allow HTML to be the legacy version HTML. HTML3 is the name of what was the HTML destination in SAS 8.2. HTML3 does not use tagsets or ODS Markup at all. It is done the old fashioned way with C-code. HTML3 has remained virtually unchanged since SAS 8.2.

Let's look at the HTML4 output first since HTML4 is the tagset that does everything. There aren't any surprises here. It looks just the way we would expect. The titles are colored, and justified on the page just the way they should be.

IMAGE MISSING....

Use this output as a reference as you look at the output for the rest of these HTML destinations. It will give you a good idea of how they all behave.

4.1.2 HTMLCSS

HTMLCSS is a basic stylesheet based HTML tagset. There is very little difference between the HTML4 tagset and HTMLCSS. Both the HTML4 and MSOffice2k tagsets inherit most of their events from the HTMLCSS tagset, and ultimately the PHTML tagset.

4.1.3 PHTML

The PHTML tagset generates a very plain HTML. The huge stylesheet is replaced by a much smaller stylesheet. If you dislike all the styles and the verbosity of the html4 and htmlcss tagsets then this tagset is a good place to start.

4.1.4 XHTML

The XHTML tagsets is one of the shortest tagsets around. It inherits from the HTML4 tagset. All it adds are the stricter XML style tags.

4.1.5 MSOffice2K

The MSOffice2k tagset is a good tagset to use if you want to import the resulting HTML into Microsoft Office. This tagset adds Microsoft specific XML to the HTML so that Microsoft office can size images better. It also creates different HTML for titles and footnotes so that they import more smoothly into Excel. This tagset also works with proc tabulate to create a better array of table cells, so that Excel will keep all of the rows straight.

4.1.6 HTML4

The HTML tagsets that use styles always use a stylesheet, if no stylesheet file is specified the stylesheet is still written to the top of the body file. Using an external stylesheet allows more flexibility and it can allow multiple reports to use the same stylesheet file.

```
ODS html file='test.html';

proc print data=sashelp.class; run;

ODS html close;
```

4.1.7 PHTML

The PHTML tagset generates a very plain HTML. The huge stylesheet is replaced by a much smaller stylesheet. If you dislike all the styles and the verbosity of the html4 and htmlcss tagsets then this tagset is a good place to start.

4.1.8 HTMLCSS

HTMLCSS is a basic stylesheet based HTML tagset. There is very little difference between the HTML4 tagset and HTMLCSS. Both the HTML4 and MSOffice2k tagsets inherit most of their events from the HTMLCSS tagset, and ultimately the PHTML tagset.

4.1.9 XHTML

The XHTML tagsets is one of the shortest tagsets around. It inherits from the HTML4 tagset. All it adds are the stricter XML style tags.

4.1.10 MSOffice2K

The MSOffice2k tagset is a good tagset to use if you want to import the resulting HTML into Microsoft Office. This tagset adds Microsoft specific XML to the HTML so that Microsoft office can size images better. It also creates different HTML for titles and footnotes so that they import more smoothly into Excel.

This tagset also works with proc tabulate to create a better array of table cells, so that Excel will keep all of the rows straight.

4.2 HTML and Excel

HTML is one of the better ways to import SAS ouput into excel. But some tagsets work better than others. HTML that uses H tags for titles, notes and bylines works better than HTML that uses tables for their formatting. That makes the HTML4 and HTMLCSS tagsets bad choices for importing into Excel. The Compact HTML, PHTML, and the MSOffice2k tagsets work the best.

4.3 CHTML

Compact HTML is a subset of HTML. Chtml has no style, it has a limited but very useful set of HTML tags. Chtml output is viewable on any browser. It is intended for use with phones and PDA's where download size is important.

4.3.1 CHTML_imode

Imode HTML is a subset of Compact HTML. Imode has no support for tables. It is primarily used for phones in Japan.

4.4 Stylesheets

If you choose to use stylesheets it can be convenient to have one stylesheet that everyone uses. It is easy to reference another stylesheet on the ODS Statement. This first ODS statement will create a stylesheet file called test.css.

```
ODS HTML file='test.html' stylesheet='test.css';
```

This next ODS statement does not create a stylesheet file at all. But it does put in a stylesheet link to the test.css stylesheet that was created earlier.

```
ODS HTML file='test.html' stylesheet=(url='test.css');
```

4.5 Javascript code

If you have a need for javascript in your HTML there are some good places to put that code. There is an event in the tagset called code_body that will write to the code file specified on the ods statement. A link to that file will be created in the body, contents, and pages files. It is also possible to trigger the code_body event when no code file is specified, that will allow support for both external and embedded javascript.

4.6 Accessibility

The HTML tagsets have accessibility features that are 508 compliant to level two. Level 3 compliance is possible with most procedures but will require updates to the table templates used by the procedures. Attributes in the table templates provide for summary, abreviation, acronym, long_description, caption, and alt. Setting these table and column attributes can improve your HTML's 508 compliance dramatically.

4.7 Summary

There are several HTML tagsets provided by SAS. Most likely one of them will be reasonably close to the style of HTML you wish to create.

Chapter 5

Using LaTeX

Besides HTML and XML, LaTeX output is one of the most useful and versatile output destinations available. LaTeX is commonly used in publishing and is capable of creating several viewable formats, including PDF and Postscript. LaTeX supports color and various image formats. Sadly, this destination is largely overlooked. This chapter will discuss how to use it to advantage.

5.1 The LaTeX statement

Latex is one of the special destination names that is shorthand for tagsets.latex. But there are other latex tagsets. Color_latex differs only in the way it invokes the usepackage statement to include the 'stylesheet' generated by latex. Simple_latex is the most simplistic form of latex which lends itself to embedding in other LaTeX documents.

The best way to use the latex tagsets is with an external stylesheet. Using an external stylesheet is the only way to turn color on. The stylesheet includes usepackage statements for all the packages needed to render the latex. It also defines macros for the styles and formatting.

Another requirements of using latex is that a url without the .sty extension be specified for the stylesheet. This is the name that will go in the usepackage statement in the preamble of the latex document. A sample ods latex statement is shown here.

```
ods latex file="test.tex" stylesheet="test.sty"(url="test");
```

5.1.1 Color Support

Color LaTeX is really the same as the regular latex tagset. The only difference is that a flag is set so that colors will be used. The entire tagset looks like this. The change is the addition of the color option.

```
define tagset tagsets.colorlatex;
    parent=tagsets.latex;

    define event stylesheet_link;
        put CR '\usepackage[color]{';
        put URL;
        put '}' CR CR;
```

```
        end;

    end;
```

A better color latex tagset would work without a stylesheet url being specified on the ods statement. This is only possible in SAS 9.1, but is a perfect example of how useful data step functions can be.

```
proc template;
    define tagset tagsets.colorlatex;
        parent=tagsets.latex;

        define event stylesheet_link;
            put CR '\usepackage[color]{';
            put scan(url, 1, '.');
            put '}' CR CR;
        end;

    end;
run;

ods tagsets.colorlatex file="test.tex" stylesheet="test.sty";
proc print data=sashelp.class;run;
ods _all_ close;
```

5.2 Compiling the LaTeX Output

There is no browser for LaTeX. LaTeX must be compiled into the document type desired. Different commands create the various forms of browseable output.

5.2.1 The latex Command

The latex command compiles LaTeX into dvi format. There are many viewers for dvi files. Dvi is not as nice as postscript or PDF but does serve as an intermediate format for postscript and other formats. DVI also does not do well with color, although the postscript generated from it will. Compiling the the output from the example above would require a statement like this.

```
latex test
```

LaTeX will create several files, all of which have different purposes. The .aux files contain measurement information. For this reason it is a good idea to run the latex command twice. The second time it will use the information it gathered the first time. The result will be better looking output. The output file from this command will be test.dvi.

5.2.2 The dvips Command

The dvi2ps command converts dvi files to postscript. The resulting postscript often looks better than the dvi output. Particularly when color is used. The following dvips command will print the contents of test.dvi to your default printer. The second command will cause the postscript to be written to a file, test.ps. There are many more options to dvips, Printer, papertype, print controls, cropping, copies, to name just a few.

```
dvips test.dvi
dvips test.dvi -o test.ps
```

5.2.3 The pdflatex Command

The pdflatex command compiles LaTeX code into PDF. This works very well and makes beautiful PDF output. The pdflatex command is used in place of the regular latex command to create pdf directly from the original LaTeX output. Like the latex command, it is a good idea to run the pdflatex twice so that measurements will be refined and used. The following command will create a file named test.pdf.

```
pdflatex test
pdflatex test
```

5.3 Integrating LaTeX output into documents

Aside from creating pdf or postscript reports it is sometimes desirable to create output that will be used in a larger LaTeX document. A paper or book for example.

5.3.1 The easy way

The easiest thing to do is write an external stylesheet, then add a usepackage statement to your LaTeX document. All of the SAS specific needs will be in that one stylesheet file. This will provide the best support for color and formatting of the output. All of the latex macros defined in that stylesheet are prefixed with 'sas' so the macro names should not clash with any latex code you already have. Using our test example from above all you need is the following line in your preamble. You can then include any parts of the ODS generated report in your own document.

```
\usepackage[color]{test}
```

5.3.2 Using NewFile to advantage

The newfile option, along with no_top and no_bot can create nicely packaged pieces of output that can be directly included into a LaTeX document. Frequently, all that is desired is the actual tabular output generated by ODS. There are several options that will help narrow the ODS output to just the parts you need. First the select and exclude statements can be used to select only the particular ODS output objects of interest. Then the ODS Markup options, newfile, no_top_matter and no_bottom_matter can be used together to create just the tables or graphs, with no surrounding latex code. The ods statement to do that will look like this.

```
ods latex file="test.tex"(notop, nobot)
          stylesheet="test.sty"(url="test")
          newfile=table;
```

The result will be a series of numbered files, starting with test.tex. All with one table per file. These files will be much easier to include into another latex document. In fact, depending on the occurance of titles and notes, these files could be left intact and included with latex's include statement, like this.

```
\include{test}
```

5.3.3 The simple way

A more simplistic method is to use the simple_latex tagset. That tagset requires nothing outside of the most basic LaTeX. This latex has a much simpler table model that uses the tabular package, and does not support color. In short, this tagset uses only the most basic latex commands to create tables. This latex tagset does not need a stylesheet so including it's output requires no usepackage to be added in the preamble of your latex document.

5.4 Image Formats and Graph

Everything would be great if the image formats attribute actually worked. But it doesn't. At least not for graph procedures. It does work for statistical graphics though. The problem for graph procedures is that no image types other than those valid for HTML are allowed. That means that png will work but not postscript. PdfLateX likes png images, but latex and dvips do not.

The fix for this is a simple tagset that converts image file extensions from .gif to .ps. Then after the job is done, the images can be replayed to the postscript device. The following example does just that.

Listing 5.1: A fix for LaTeXand graph

```
proc template;
   define tagset tagsets.mylatex;
      parent = tagsets.latex;

      image_formats = 'ps,psepsf,png,jpeg,gif';

       define event image;
           put '\sasgraph{';
           put BASENAME / if !exists(NOBASE);

           /* convert gif extension to ps.          */
           /* use eps if you use the psepsf driver. */
           put tranwrd(URL, 'gif', 'ps');

           put '}' CR;
       end;
   end;
run;

filename junk ".";

ods tagsets.mylatex file="graph.tex";
goptions dev=gif target=ps;

proc gchart data=sashelp.class;
vbar age / pattid=midpoint;
run;
quit;

proc gplot data=sashelp.class;
plot height*weight;
```

```
run;
quit;

ods _all_ close;

/*-------------------------------------------------------------eric-*/
/*-- Replay the graphs to generate postscript.                  --*/
/*-------------------------------------------------------------16Oct03-*/
goptions dev=ps gsfname=junk;
proc greplay nofs;
   igout work.gseg;
   replay _all_;
run;
quit;
```

5.5 LaTeX in the different versions of SAS

5.5.1 SAS 8.2

While ODS Markup was experimental in SAS 8.2 it was still a fairly stable product. The biggest exception to that was the LaTeX output. The Embedded_stylesheet option was not an available feature in ODS Markup at in that release. An external stylesheet had to be used.

But when a stylesheet was specified, a crash occured. Because of tagsets, SAS was able to work around this problem by making a new latex tagset available. The new tagset was called latex2. The latex2 tagset solved the problem by writing a fixed set of style definitions to the preamble of the LaTeX document. No stylesheet option was needed. The disadvantage was that the ods markup style option had no effect. Using this latex in other documents was also very problematic.

5.5.2 SAS 9.0

In SAS 9.0 LaTeX worked much better. The LaTeX code itself was much cleaner and more adaptable. A common complaint was using the output as inclusions in other documents. One of the changes was the addition of the 'sas' prefix on all of the ODS generated latex macros.

5.5.3 SAS 9.1 and beyond.

LaTeX has continued to evolve with the addition of the simple Latex tagset. The latex output is more versatile than ever.

Another addition in SAS 9.2 is measured markup destination. The measured markup destination adds measurement variables to tagsets. This destination is more or less invisible, since the behavior is turned on by the tagset's measurement attribute. This is the same sort of measurement that the ODS RTF and Printer destinations do. The first goal of this destination is support an RTF tagset. But LaTeX will also benefit from this.

Turning on measurement will enable the LaTeX tagset to make intelligent decisions about table panelling and paging.

5.6 Summary

LaTeX is one of the most useful outputs that ODS creates. It can be easily be used within Documents of any size from reports and papers to books. It can be used to create many output types including PDF, Postscript, and DVI.

Part II

Beginning Tagsets

Chapter 6

Tagsets: how do they work?

Tagsets are a hybrid of procedural and non-procedural, declarative programming languages. A tagset is a collection of event definitions. Each definition can have procedural elements, but the events themselves are not procedural. That's why they are called events. Events do happen in some order, but with variation. They are like the events in your day. The alarm clock, brushing your teeth, eating breakfast, going to work, etc. They do have some predictability, but are not exactly the same all the time. In the context of SAS, events are determined by the job being run. There are events for titles and footers, and page breaks. There are events for tables and images, Tables have varying numbers of observations, columns, and headers. There are events for all of those things and more. This chapter will use a simple plain text tagset to show how tagsets work. As the tagset evolves the output created will reveal the fundamentals of tagset behavior.

6.1 Data and context in time

Tagsets are an event driven programming language. While they do have procedural elements the overall structure is an event driven model. Events are best described as named bundles of data that occur at different points in time. The name and timing of of the event provide context to the data it contains.

6.2 Event Requests

A tagset defines what is to be done with the data for any given event. But it is not necessary to define all possible events. Only events which are meaningful to the output being created need to be defined. For this reason, it is more accurate to think of these data bundles as event requests. You may have a title statement in your SAS job. When it comes time, ODS will bundle up all the data, and metadata which defines that title. The title event will then be requested. If the tagset in use has a definition for the title event then something will happen. Otherwise, nothing will happen.

6.2.1 A few variables

At current count there are 279 variables that could be defined for any given event. Most events only populate a fraction of the available variables. None populate all of them at once.

Data and Metadata

Of those 279 variables, 172 of them are concerned with the data and the metadata about the data. The main variable for any event is called 'value'. All of the other variables are concerned with providing more information abouth that value. Name, label, type, missing, rawvalue, justification, to name a few. These variables are defined by ODS, the procedure currently in use, and the table or statgraph template for the current object.

Still other variables provide context within the output, the procedure name, the output count, the row, count, the date, and many more.

style Variables

The remaining 107 variables are style variables. These are the same as the attributes you may have defined in the ODS style template that you are using. These variables define things like font, foreground color, background color, cellspacing, bordercolor, and many more.

6.3 Our First tagset

It's best not to get bogged down in the details, or to spend much time worrying about how all this works. Tagsets are self revealing, the best way to learn them is to play with them. Starting with something simple is the best way to go. It will all make sense in time.

This series of examples is going to create a very simple tagset. The output it will create will be plain text, something like the listing output that SAS has generated forever.

6.3.1 Define statement

The first thing to do is give a name to the new tagset. That is done with the define statement. Our tagset is going to be called plain_text. The tagset should be put in proc template's tagset directory, so it's first name is *tagsets.* . The code in listing 6.1 shows a very simple tagset definition.

Listing 6.1: Defining a Tagset

```
Define tagset tagsets.plain_text;

end;
```

6.3.2 The data event

The data event is one of the most fundamental events. This event is triggered for every value in every observation of a table. The define event statement is used to create an event definition. The code now looks like this 6.2.

Listing 6.2: Adding an Event

```
Define tagset tagsets.plain_text;

    Define event data;
```

```
        end;

    end;
```

The put statement

So far so good but the tagset doesn't do anything yet. The values in the data event need to be printed The put statement is the way to do that. If you are familiar with data step, this may seem familiar but it is not quite the same as the put statement in Data Step. This put automatically concatenates everything. If you want a newline you must specify it with nl.

```
Define tagset tagsets.plain_text;

    Define event data;
        put 'Data: ' value nl;
    end;

end;
```

6.3.3 The header event

The data will now be printed but there will be no headings. The header event will take care of that. All that is needed is a header event that looks just like the data event.

For now, printing each value on a line by it's self is enough. The working tagset, along with a simple proc print is shown in figure 6.3 on page 35 The plain text output is shown in figure 6.4 on page 36 The corresponding latex output is shown in figure 6.3.3 on page 36 Notice that the observation count on each row is handled by the header event.

Listing 6.3: A Plain Text Tagset

```
proc template;

    Define tagset tagsets.plain_text;

        Define event data;
            put 'Data: ' value nl;
        end;

        Define event header;
            put 'Header: ' value nl;
        end;


    end;

run;
```

```
ods latex file='test.tex' stylesheet="test.sty" (url="test");
ods tagsets.plain_text file='test.txt';
option obs=3;
proc print data=sashelp.class; run;
ods tagsets.plain_text close;
ods latex close;
```

Listing 6.4: Output: Simple Tagset Output

```
Header: Obs
Header: Name
Header: Sex
Header: Age
Header: Height
Header: Weight
Header:  1
Data: Alfred
Data: M
Data: 14
Data: 69
Data: 112.5
Header:  2
Data: Alice
Data: F
Data: 13
Data: 56.5
Data: 84.0
Header:  3
Data: Barbara
Data: F
Data: 13
Data: 65.3
Data: 98.0
```

The SAS System

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

6.4 Fleshing out the plain_text tagset

This is all very nice but to create something more useful some more events will be needed. The most basic events are fairly intuitive. All tables have a handful of basic events. We've seen the data and header events. The other basic events are, table, row, table_head, table_body, and table_foot.

The table event encapsulates all other events which define a table. You may be wondering how an event can contain other events within it's scope, Or even how an event can have a scope.

Start and finish

Event requests commonly come in matched pairs. If you are used to coding HTML then this is a common concept for you. Most HTML tags are paired as a begin tag and and end tag such as and . Tagset events are the same way. But to keep things neatly packaged each event definition can hold both parts. The beginning and the ending. In the tagset these are denoted by start: and finish:.

This works because most events get called twice from within ODS. Once at the beginning and then again at the end. So when print procedure creates a table, the table event is triggered twice, once at the begininng of the table, and once at the end. The same thing goes for the row event, the table_head, table_body, and table_foot events, even the data and header events.

Our output doesn't look much like a table, but now there are some more events to try out. A slight modification to the data and header event and the addition of the table and row events will make the output look much nicer. The new tagset now looks like the code in figure 6.5 on page 37

Listing 6.5: Simple Tagset

```
proc template;

    Define tagset tagsets.plain_text;

        Define event table;
            start:
                put '-------------------------------------------------' nl;
            finish:
                put '-------------------------------------------------' nl;
        end;

        Define event row;
            finish:
                put nl;
        end;

        Define event data;
            put value '   ';
        end;

        Define event header;
            put value ' ';
        end;


    end;

run;


ods tagsets.plain_text file='test.txt';
option obs=3;
proc print data=sashelp.class; run;
ods tagsets.plain_text close;
```

Listing 6.6: Output: Simple Tagset Output

```
Obs Name Sex Age Height Weight
 1 Alfred   M   14    69   112.5
```

This version isn't half bad considering the simplicity of the tagset. The table event puts lines above and below, the finish of row event causes a newline at the end of each row, and the data and header values are printed with spaces between them.

6.4.1 head, body, and foot

The table_head, table_body, and table_foot events can also be useful to us. The head event surrounds the rows that contain the table headings. The body section surrounds the data rows, and in the rare cases that the table has footers at the bottom, the foot section surrounds those.

For explanation this new tages is going to use all of these events so you can see where they fit in. The output will be a bit garish but that can be fixed later. Our new tagset is shown figure 6.7 on page 38 And the output, expanded to 3 observations is in figure 6.8 on page 39

Listing 6.7: A Simple Tagset

```
proc template;

    Define tagset tagsets.plain_text;

        Define event table;
            start:
                put '-----------------------------------------' nl;
            finish:
                put '-----------------------------------------' nl;
        end;

        Define event table_head;
            finish:
                put '_________________________________________' nl;
        end;

        Define event table_body;
            start:
                put '.........................................' nl;
            finish:
                put '.........................................' nl;
        end;

        Define event table_foot;
            start:
                put '=========================================' nl;
            finish:
                put '=========================================' nl;
        end;
```

```
        Define event row;
            finish:
            put nl;
        end;

        Define event data;
            put value '   ';
        end;

        Define event header;
            put value ' ';
        end;


    end;

run;


ods tagsets.plain_text file='test.txt';
option obs=3;
proc print data=sashelp.class; run;
ods tagsets.plain_text close;
```

Listing 6.8: Output: Simple Tagset Output

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

It is worth noting that the table foot event did not create any output. That is because there was no table foot. If there had been something inside the table's foot section then that group of events would have generated output.

6.4.2 Titles

There is only one thing that the latex output in figure 6.3.3 on page 36 has that the plain_text output doesn't print the default system title. Only one more event is needed for that. But there are other events related to the system_title event that are worth noting. There are actually 8 more fundamental events which are concerned with titles, and notes. The names are self explanatory. They are, System_title, System_footer, Proc_title, Byline, Note, Warning, Error, and Fatal. Like the incredibly simple data and header events all that is really needed for each of these is to print the value.

The new tagset is shown figure 6.9 on page 40 And the output is in figure 6.10 on page 41

Listing 6.9: A Better Simple Tagset

```
proc template;

    Define tagset tagsets.plain_text;

        Define event table;
            start:
                put '--------------------------------------------------' nl;
            finish:
                put '--------------------------------------------------' nl;
        end;

        Define event table_head;
            finish:
                put '__________________________________________________' nl;
        end;

        Define event row;
            finish:
            put nl;
        end;

        Define event data;
            put value '   ';
        end;

        Define event header;
            put value ' ';
        end;

        Define event system_title;
            put value nl nl;
        end;

        Define event system_footer;
            put value nl nl;
        end;

        Define event proc_title;
            put value nl nl;
        end;

        Define event note;
            put value nl nl;
        end;

        Define event warning;
            put value nl nl;
        end;
```

```
        Define event error;
            put value nl nl;
        end;

        Define event fatal;
            put value nl nl;
        end;

    end;

run;

ods tagsets.plain_text file='test.txt';
option obs=3;
proc print data=sashelp.class; run;
ods _all_ close;
```

Listing 6.10: Output: Simple Tagset Output

```
The SAS System

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

6.5 Summary

By giving away a few events, this chapter has given a glimpse of how tagsets and events work. Events can have a start and finish, and can occur within the scope of another event. The way the data event occurs between the start and finish of the row event.

Tagsets can be very simple and so can their events. This can be used to advantage. It is possible to actually write very simple tagsets to explore how they work. This is a fundamental concept behind learning, understanding, and using tagsets.

Chapter 7

Modifying Existing Tagsets

The most common tagset programming is done to modify the behavior of an existing tagset. Modifying a tagset is fairly easy to do because tagsets can inherit events from each other. It is also easy to identify which events need changing. This chapter will introduce a few new concepts while showing the steps needed to modify a tagset through inheritance.

7.1 Changing the delimiter for CSV files

It is a common request to change the delimiter used by the CSV destination. A common alternative to the comma is the semi-colon, another popular choice is tab. These examples are going to show how to create a new tagset, that does this through inheritance. The CSV tagset is not as simple as it once was. The difference between the current CSV tagset and the original CSV tagset that shipped with SAS 8.2 is like night and day. Despite it's surprising complexity the CSV tagset is still easy to derive other tagsets from.

With the version from SAS 8.2, creating a tagset that used a different delimiter was actually more complex. The current version actually has an option that we can specify on the ODS statement or within a new tagset that we define ourselves.

Because it is a good example, I will take you through the process of modifying the CSV tagset from SAS 8.2. If you need it this tagset is readily available at http://support.sas.com/rnd/base/topics/odsmarkup.

After that, we'll go through the motions with the CSV tagset available for SAS 9.1.3 dated April 2006.

7.1.1 Finding the Events

The first thing to do is find the events that put the commas in between the comma separated values. A good guess is that the data and header events will be involved. But one of the nice things about inheriting from another tagset is that it can be examined for clues. Since this tagset is going to modify the csv tagset, that is the tagset to inherit from. There are two ways to look at the csv tagset. Proc template is my least favorite, because it loses white space and comments. It also reverses the order of the tagset attributes and events. But to use it do the following.

Listing 7.1: Getting the CSV Source

```
proc template;
```

```
source tagsets.csv;
```

My favorite way to look at tagsets is by using the original source which is downloadable from the SAS web site. The source can also be examined in the template browser or by using the source statement in proc template. However, those methods do not show white space or comments that are in the original source. That makes the original tagset source code far superior to the tagset source you can see through the template interface.

Looking at the source in Figure 7.2 on page 44 shows all the events which have commas.

Listing 7.2: CSV Events with commas

```
define event header;
  start:
    put ',' / if !cmp(COLSTART, "1");
    put '"';
    put VALUE;
  finish:
    put '"';
end;

define event data;
  start:
    put ',' / if !cmp(COLSTART, "1");
    put '"';
    put VALUE;
  finish:
    put '"';
end;

define event colspanfill;
    put ',';
end;

define event rowspanfill;
    put ',' /if ! exists(VALUE);
end;
```

7.1.2 Simple Ifs

These events are readily understandable, but contain three if statements that need explanation. This type of if only applies to the statement it is connected to. There are 5 tests that are extremely efficient. The first two are, cmp() and exists(). The others are any(), contains(), and a variable by its self. Exists, Any, and the simple variable tests evaluate for a non-zero length of strings and the actual value of numeric variables. A value of 0 is false, non-zero is true.

Compare

Both of these tests are preceded by an exclamation point.[1] An exclamation point or caret means not. Cmp is an effient case insensitive compare. The variable colstart always contains the number of the column that the current cell starts in.

The data and header events, use cmp to suppress the comma, on first value of the row.

Exists

The final if in the tagset is an exists. Exists can take any number of variables as arguments. If all the variables have a value then exists is true, otherwise it is false. The following pseudo code shows how exists can be thought of as a group of ands.

```
if exists(variable1, variable2, variable3)
if variable1 and variable2 and variable3
```

In a case like this, the variable could be listed by its self like this.

```
put ',' /if ! exists(variable);
put ',' /if ! variable;
```

Either syntax will cause the put in the rowspanfill event will print a comma if the VALUE variable does not have a value. Rowspanfill is one of those elusive events that is is rarely sighted. The best way to coax it out of hiding is to use Proc Tabulate.

Any

Any() is just like exists() but is true if any one of the variables given has a value. Think of it as a group of OR's as demonstrated by this pseudo code.

```
if any(variable1, variable2, variable3)
if variable1 or variable2 or variable3
```

7.2 Making the changes

Even with the new syntax just discussed, the job is pretty easy. All that is desired is to change those commas to semi-colons.

Tagset inheritiance is very straight forward. A tagsets inherits all attributes and events from the parent tagset. Any changes require a complete redefinition of the event to be changed.

The new tagset only needs to specify tagsets.csv as it's parent and redefine those four events with semi-colons instead of commas. The new tagset is shown in listing 7.3.

Listing 7.3: Semi-Colon separated values

```
Define tagset tagsets.SSV
    parent=tagsets.csv;

    define event header;
        start:
```

[1] An ! can only occur at the beginning of a test

```
            put ';' / if !cmp(COLSTART, "1");
            put '"';
            put VALUE;
        finish:
            put '"';
    end;

    define event data;
        start:
            put ';' / if !cmp(COLSTART, "1");
            put '"';
            put VALUE;
        finish:
            put '"';
    end;

    define event colspanfill;
        put ';';
    end;

    define event rowspanfill;
        put ';' /if ! exists(VALUE);
    end;

end;
```

Running the following proc print with the new tagset gives us the output. Shown in figure 7.4.

```
options obs=3;
ods tagsets.ssv file='test.ssv';
proc print data=sashelp.class;run;
ods tagsets.ssv close;
```

Listing 7.4: Output: Semi Colon Separated output

```
"Obs";"Name";"Sex";"Age";"Height";"Weight"
" 1";"Alfred";"M";"14";"69.0";"112.5"
" 2";"Alice";"F";"13";"56.5";" 84.0"
" 3";"Barbara";"F";"13";"65.3";" 98.0"
```

7.3 A better CSV tagset

This new but old tagset is nice, but it's not very flexible. It's not that hard to write a new one if a different delimiter is desired, but wouldn't it be nice to have a tagset that would use any delimiter it was given? Macro variables are one way to achieve more flexibility. Once declared they can be used anywhere within the tagset. Macro variables in a tagset can get out of hand. Since SAS 9.1.3 there has been an options option that gives the same flexibility without cluttering up the SAS macro name space.

7.3.1 Macro variables

Macro variables are one way to pass values into a tagset. Any macro variable can be accessed within the tagset as long as it is declared. A macro variable can be used to define the delimiter. A comma can be used in its absence. A macro variable is declared in a tagset like this:

```
mvar myvariable;
nvar myOtherVariable;
```

As long as they are declared the tagset will know what to do when they are used. If they are not declared then the tagset will fail to compile when you reference them later.

7.3.2 The Initialize Event

The code could use the macro variable everywhere there is a comma, but that would require an if at each point to determine if a comma was needed instead. That would add unecessary complexity. It would also allow the delimiter to be changed in midstream. A changing macro variable might be good in other situations, but it's not very useful here.

It would be best to set up the delimiter once at the very beginning and use that everywhere. There is a special event especially for this sort of thing. The Initialize event[2] happens once when the tagset is first invoked. All that is needed is to examine the macro variable and create a tagset variable accordingly.

7.3.3 The set statement

By using the Set statement[3] it is possible to create variables within a tagset. All variables are global and will persist until they are unset or the output destination is closed. Set will be used to create a new delimiter variable that will be populated with the macro variable or, lacking that, a comma.

7.3.4 The New CSV Tagset

Our new old school (SAS 8.2) tagset is shown in figure 7.5.

Listing 7.5: A Better CSV Tagset

```
proc template;

    Define tagset tagsets.SSV;
        parent=tagsets.csv;

        mvar csv_delimiter;

        define event initialize;
           set $delimiter csv_delimiter;
           set $delimiter ',' /if !$delimiter;
        end;

        define event header;
            start:
```

[2]Only available in SAS 9.0 or greater, use the doc event in earlier releases
[3]Available in SAS 9.0 and greater

```
                put $delimiter / if !cmp(COLSTART, "1");
                put '"';
                put VALUE;
            finish:
                put '"';
        end;

        define event data;
            start:
                put $delimiter / if !cmp(COLSTART, "1");
                put '"';
                put VALUE;
            finish:
                put '"';
        end;

        define event colspanfill;
            put $delimiter;
        end;

        define event rowspanfill;
            put $delimiter /if ! exists(VALUE);
        end;

    end;
run;

%let csv_delimiter=|;

options obs=3;
ods tagsets.ssv file='test.ssv';
proc print data=sashelp.class;run;
ods tagsets.ssv close;
```

For this run the delimiter has been set to '|' which causes the output to appear as in figure 7.6 on page 48

Listing 7.6: Output: Semi Colon Separated output

```
"Obs"|"Name"|"Sex"|"Age"|"Height"|"Weight"
" 1"|"Alfred"|"M"|"14"|"69.0"|"112.5"
" 2"|"Alice"|"F"|"13"|"56.5"|" 84.0"
" 3"|"Barbara"|"F"|"13"|"65.3"|" 98.0"
```

This tagset is quite flexible. But there's the next best thing that will make it even more flexible. Macro variables are nice, but they can be hard to keep track of.

7.3.5 Tagset Alias

At some point there was a realization that a tagset might want to behave differently on occasion. That it would assume an alias, a sort of alternate personality. from it's usual behavior. What I didn't realize at that

time is that the future would call for a multiply split personality and that a simple alias would fall quite short of providing the nuances we would later need.

The ods markup statement has an alias option that was created just for this sort of purpose. It works very simply, any value assigned to alias becomes the value of the tagset_alias variable in the tagset. All that is needed is to add it to the initialize event. Once that happens, the delimiter can be specified on the ods statement.

For simplicity this example creates a completely new tagset which inherits from the example in 7.5 on page 47. The only event that needs to be changed is the initialize event. There is no reason to repeat all the other events. Inheritance keeps the example short and simple. In reality this would probably be incorporated into one tagset.

The delimiter should be the value specified on the ods statement first. The value of the macro variable second, and if neither of those are specified then the delimiter should default to comma. The resulting code is shown in figure 7.9 on page 50. In this example the macro variable is still set to '|' but an ! is given on the ods statment. The resulting output is shown in figure 7.8 on page 49.

Listing 7.7: Using alias

```
proc template;

    Define tagset tagsets.SSV2;
        parent=tagsets.ssv;

        mvar csv_delimiter;

        define event initialize;
           set $delimiter tagset_alias;
           set $delimiter csv_delimiter /if !$delimiter;
           set $delimiter ',' /if !$delimiter;
        end;
     end;
run;

%let csv_delimiter=|;

options obs=3;
ods tagsets.ssv2 file='test.ssv' alias='!';
proc print data=sashelp.class;run;
ods tagsets.ssv2 close;
```

Listing 7.8: Output: Variable delimited output

```
"Obs"!"Name"!"Sex"!"Age"!"Height"!"Weight"
" 1"!"Alfred"!"M"!"14"!"69.0"!"112.5"
" 2"!"Alice"!"F"!"13"!"56.5"!" 84.0"
" 3"!"Barbara"!"F"!"13"!"65.3"!" 98.0"
```

7.3.6 Tagset Options

As of SAS 9.1.3 the ODS markup destination has a new option. Options can now be passed to the tagset with the options(...) option. This allows great flexibility in the tagset with a central control in the ods statement.

Options are much more powerful than the Alias option, and much easier than macro variables. As a tagset becomes more complex it is often necessary to give it more controls. Macro variables can get out of hand and become hard to manage. As of this writing, the most current CSV tagset has several options. Other tagsets have even more.

This new tagset, in Figure 7.9 on page 50, is a great place to start when creating any new tagset. As new options are added the help can be updated at the same time. All the parts are here to create options and documentation as you go.

```
ods tagsets.ssv2 file='test.ssv' options(delimeter='!' doc='help');
```

Any name/value pairs given within the parenthesis become entries in a tagset dictionary variable called $options. It is up to the tagset to use them. It is also the responsibility of the tagset to document their use.

Listing 7.9: Using alias

```
proc template;

    Define tagset tagsets.SSV2;
        parent=tagsets.ssv;

        mvar csv_delimiter;

        define event initialize;
           set $delimiter tagset_alias;
           set $delimiter $options['DELIMITER'] /if $options;
           set $delimiter csv_delimiter /if !$delimiter;
           set $delimiter ',' /if !$delimiter;

           trigger help /if $options;
        end;

        define event options_set;
           trigger initialize;
        end;

        define event help;
           break /if ^cmp($options['DOC'], 'help');
           putlog ' ';
           putlog 'Help for the SSV2 tagset';
           putlog ' ';
           putlog 'The following options are available';
           putlog 'doc: No default value.';
           putlog '     help - this text.';
           putlog ' ';
           putlog "delimiter: default value: ','";
           putlog '     The value of this option will be used';
           putlog '     as the delimeter for the data printed.';
           putlog ' ';
        end;

     end;
run;
```

```
%let csv_delimiter=|;

options obs=3;
ods tagsets.ssv2 file='test.ssv' options(delimiter='!' doc='help');
proc print data=sashelp.class;run;
ods tagsets.ssv2 options(delimeter=';');
proc print data=sashelp.class;run;
ods tagsets.ssv2 close;
```

This tagset introduces a new event called Options_set. The Options_set event will happen when a subsequent ods statement changes the options on an ods statement that is already open. The second ods statement in the example exercises the Options_set event.

7.4 The Current Methodology

All of these examples have been great for getting an understanding of the CSV tagset, events, if statements, macro variables, Tagset Alias, and the Options option. But how would you do all of this now? With the CSV tagset in all of it's modern complexity. Well, that's not so hard as all of this.

In it's current version, the CSV tagset has an option for the delimiter. And another for turning on titles, and another for footnotes, bylines, procedure titles, and notes. Even for turning column headers on and off or changing the way currency values are treated. That can lead to a rather long set of options on the ods statement.

What if you wanted CSV output that had titles but no table headers, and it treated money like numbers. And we wanted a semi-colon as a field separator. Here is the ods statement we could use.

```
ods csv file='test.csv'
        options(delimiter=';' titles='yes' Currency_as_number='yes');
```

The interesting thing about this is that there are other CSV tagsets. CSVALL and CSV_byline don't really do much now. With SAS 8.2 they actually had a fair amount of code in them. Currently all they do is set up some options. They are a great example of how we can create a new destination that behaves quite differently from the tagset it was derived from. But without adding much code at all.

Here is the entire source code for the CSVALL tagset.

```
define tagset tagsets.csvall;
    parent= tagsets.csv;
    notes "This is the CSV with titles and bylines definition";

    define event initialize;
        set $options['BYLINES'] 'yes';
        set $options['TITLES'] 'yes';
        set $options['PROC_TITLES'] 'yes';
        set $options['NOTES'] 'yes';
        trigger set_options;
        trigger documentation;
        trigger compile_regexp;
    end;

end;
```

This can be used as a template for other CSV tagsets. To create this tagset the initialize event was copied from the parent CSV tagset. Set statements were added to create a tagset that would over ride any settings or lack of settings on the ods statement. In this case the options for bylines, titles, proc titles and notes are all turned on. It would be just as simple to create a new tagset that changed the default delimiter.

In this example we want to test for an incoming value before setting it to something else. That is actually something that might be nice in the CSVALL tagset. As it stands, the CSVALL tagset would not allow for the titles to be turned off. It would set that option to yes regardless of the options on the ods statement. Here is the new ssv tagset.

```
define tagset tagsets.ssv;
    parent = tagsets.csv;
    notes "This is the CSV with titles and bylines definition";

    define event initialize;
        do /if ^$options['DELIMITER'];
            set $options['DELIMITER'] ';';
        done;
        trigger set_options;
        trigger documentation;
        trigger compile_regexp;
    end;

end;
```

I think you'll agree that tweaking tagset behavior has become easier although the tagsets themselves are more complex. It is important to be mindful of the future when writing tagsets. Compartmentalization makes future enhancements much easier.

7.5 Summary

All of the things covered here will be covered in more detail later. These few examples have covered a lot of ground.

Most of the idea's introduced here are used extensively in all tagsets. Inheritance is the primary way of modifying and creating new tagsets. Simple if statements are the most common form of controlling the program flow within events. Macro variables, tagset alias, and the options option are ways to affect the personality of a tagset from one usage to the next.

Chapter 8

The Path to Enlightenment

So far all of the examples have contained only the events and variables needed. But the ability to find events, and their variables is required to be truly effective at programming tagsets. This chapter will explain several techniques that will illustrate which events may work for a solution, and the variables that are bundled in those events. In short, this chapter will show how to get tagsets to reveal themselves.

8.1 Finding Events

So far, to keep things simple, a few event names have been explained as though they are the only events needed. But there are many more events than those few that have been revealed. Like the events in your day, the event requests generated by ODS also vary. It all depends on the job being run and which output file is examined. The events for the body file will be different than the contents file which will be different from the stylesheet file. A job with bygrouping will be different than a job without it. Footnotes, graphs, notes, the procedure, all of these things cause variations in the event requests that ODS makes. Similarly, a job with By group processing will be different than a job without it. A Proc Print job with By group processing will also be different from other procedures that also do By group processing, like Proc Tabulate or Proc Means.

8.1.1 The Short_map Tagset

Thankfully a tagset can be written that will tell what the events are and when they occur. In fact there are several tagsets already created for this purpose. All but one of these tagsets generate XML. Where the tag names are the event names. There is no need to worry if you don't know anything about XML. The output will become self evident when you see it. The main difference between these tagsets are the attributes displayed on the tags. The Super_map tagset is the newest and most capable mapping tagset. Although it may take a little learning to get up to speed. By default Super_map behaves just like Short_map. Short_map output is the simplest and easiest to understand of all the mapping tagsets. The Event_map tagset is fairly verbose and displays the most attributes. Short_map shows only the values, names, and labels and is more than adequate most of the time. Tpl_style_map is more concerned with style names, and the names for ODS output objects. Text_map is meant as a learning tool but is not as helpful as the others once events are understood. The mapping tagsets are shown in table 8.1 on page 54.

Using the short_map tagset without running a procedure is a good first start. All that is needed is two ods statements as shown in figure 8.1 on page 54. The corresponding output is shown in figure 8.2 on page

Table 8.1: Event Mapping Tagsets

Super_map	Extremely versatile, it has the same abilities as event_map and short_map as well as regular expression mat
Event_map	A fairly verbose XML map
Short_map	The same Event_map but less data
tpl_style_map	Displays more style and template information
text_map	A formatted text map. More human readable than xml
odsxrpcs	The most verbose of all tagsets. All attributes are shown as tags.

54. What is important about the output is that all of the events shown are the result of the ODS invocation and close.

Listing 8.1: Creating an Empty Event Map

```
ods tagsets.short_map file='test.xml';
ods tagsets.short_map close;
```

Each tag in this output is the name of an event. Using standard XML format a the <doc> tag corresponds to the Start: section of the doc event. The </doc> corresponds to the Finish: section of the doc event. The <doc_title/> tag means that this is a non-scoping event that does not really have a start and finish. If the event is defined with a start: and finish: that is still ok, but both sections will be triggered one after the other when the event occurs.

It's interesting to note that even though the job didn't run a procedure there were still a fair number of events that occured. Markup languages generally need a head section at the top of the document which has information about the document itself. This section also holds information about how the document should be formatted. The very last events are doc_body and the end of doc. Doc_body is where the output from our procedures will go.

It is worth noting that the initialize event did not show up in the map. We already know it is the first event that happens. That is because the short_map tagset specifically omits it.

The excercises in the next chapter are also very informative. Particularly the example which specifies all possible files on the ODS Markup statement on page 80.

Listing 8.2: Output: Empty Event Map Output

```
<doc operator="eric" sasversion="9.1"
     saslongversion="9.01.01M0D10062003"
     date="2003-10-24" time="18:09:16" encoding="iso-8859-1">
  <doc_head>
    <doc_meta/>
    <auth_oper/>
    <doc_title/>
    <stylesheet_link/>
    <javascript>
      <startup_function>
      </startup_function>
      <shutdown_function>
```

```
      </shutdown_function>
    </javascript>
  </doc_head>
  <doc_body>
  </doc_body>
</doc>
```

The output becomes much more interesting if the job is modified to run the print procedure with one observation. As expected there will now be table, row, header and data events. But there are also a lot of other new events.

Listing 8.3: Simple Event Map Output

```
options obs=1;
ods tagsets.short_map file='test.xml';
proc print data=sashelp.class; run;
ods tagsets.short_map close;
```

The output is rather extensive for such a small table!

```
<?xml version="1.0" encoding="iso-8859-1"?>

<doc operator="eric" sasversion="9.1"
     saslongversion="9.01.01M0D10062003"
     date="2003-10-24" time="18:27:50" encoding="iso-8859-1">
  <doc_head>
    <doc_meta/>
    <auth_oper/>
    <doc_title/>
    <stylesheet_link/>
    <javascript>
      <startup_function>
      </startup_function>
      <shutdown_function>
      </shutdown_function>
    </javascript>
  </doc_head>
  <doc_body>
    <proc name="Print">
      <anchor name="IDX"/>
      <page_setup>
        <system_title_setup_group>
          <title_setup_container>
            <title_setup_container_specs>
              <title_setup_container_spec/>
            </title_setup_container_specs>
            <title_setup_container_row>
              <system_title_setup value="The SAS System">
              </system_title_setup>
            </title_setup_container_row>
          </title_setup_container>
        </system_title_setup_group>
```

```
</page_setup>
<system_title_group>
  <title_container>
    <title_container_specs>
      <title_container_spec/>
    </title_container_specs>
    <title_container_row>
      <system_title value="The SAS System">
      </system_title>
    </title_container_row>
  </title_container>
</system_title_group>
<proc_branch name="Print" value="Print">
  <leaf name="Print" value="Data Set SASHELP.CLASS">
    <page_anchor/>
    <output name="Print">
      <table>
        <rowspec>
          <cellspec/>
          <cellspecsep/>
          <cellspec/>
          <cellspecsep/>
          <cellspec/>
          <cellspecsep/>
          <cellspec/>
          <cellspecsep/>
          <cellspec/>
          <cellspecsep/>
          <cellspec/>
        </rowspec>
        <colspecs>
          <colgroup>
            <colspec_entry name="Obs"/>
          </colgroup>
          <colgroup>
            <colspecsep/>
            <colspec_entry name="Name"/>
            <colspecsep/>
            <colspec_entry name="Sex"/>
            <colspecsep/>
            <colspec_entry name="Age"/>
            <colspecsep/>
            <colspec_entry name="Height"/>
            <colspecsep/>
            <colspec_entry name="Weight"/>
          </colgroup>
        </colspecs>
        <table_headers>
          <header_spec>
            <sub_header_colspec name="Obs">
```

```
        <col_header_label value="Obs">
        </col_header_label>
      </sub_header_colspec>
      <sub_header_colspec name="Name">
        <col_header_label value="Name">
        </col_header_label>
      </sub_header_colspec>
      <sub_header_colspec name="Sex">
        <col_header_label value="Sex">
        </col_header_label>
      </sub_header_colspec>
      <sub_header_colspec name="Age">
        <col_header_label value="Age">
        </col_header_label>
      </sub_header_colspec>
      <sub_header_colspec name="Height">
        <col_header_label value="Height">
        </col_header_label>
      </sub_header_colspec>
      <sub_header_colspec name="Weight">
        <col_header_label value="Weight">
        </col_header_label>
      </sub_header_colspec>
    </header_spec>
  </table_headers>
  <table_head>
    <row>
      <header value="Obs">
      </header>
      <header value="Name">
      </header>
      <header value="Sex">
      </header>
      <header value="Age">
      </header>
      <header value="Height">
      </header>
      <header value="Weight">
      </header>
    </row>
  </table_head>
  <table_body>
    <row>
      <header value=" 1">
      </header>
      <data value="Alfred">
      </data>
      <data value="M">
      </data>
      <data value="14">
```

```
                    </data>
                    <data value="69">
                    </data>
                    <data value="112.5">
                    </data>
                  </row>
                </table_body>
              </table>
            </output>
          </leaf>
        </proc_branch>
      </proc>
    </doc_body>
  </doc>
```

That is a lot of output. Obviously the probability of using all of those events in one tagset is pretty slim. But for every event there is probably a tagset that uses it.

8.2 The Default_Event Tagset Attribute

So how do you know that you are seeing all the events? The answer is because these tagsets use the default event. Normally, if an event is not defined, the event request is quietly forgotten. But a tagset can have an event that is designated as the default. If an event is not found, in a tagset that has a default event, the default event is triggered instead. The code for the shortmap tagset can be seen in figure 8.4 on page 59.

The first thing is a bunch of attributes at the top of the tagset. The attribute to look for is the default_event. In this case the default event is set to 'basic'.

8.2.1 The Events

The first event is the initialize event. This empty definition ensures that the initialize event request will be quietly dropped.

The show_charset event sets $show_charset variable if suppress_charset is not set. Suppress_charset is a setting that can be made in the SAS registry. This is the tagset's way of recognizing it.

The doc event is defined almost the same as the basic event but prints some extra information that is good to know.

The doc event also does something we haven't seen yet. It triggers the show_charset event. Trigger allows the creation of arbitrary event definitions, that can be used as if they are functions. Triggers are extremely efficient. At run time they behave as if the triggered event's statement's were right where the trigger statement is. Similar to, but not quite the same as SAS macro's or inline functions in lower level programming languages like 'C'.

The next new statement is putq. Putq works just like put, but it automatically double quotes any variables it prints. This is perfect behavior for printing HTML or XML attributes.

After printing the the xml tag, The doc event creates the doc tag by printing an < followed by the event_name variable. Event_name will always have the name of the initial event requested from within ODS. In this case the value will be 'doc'.

The rest of the start section is nothing but putq's. Looking at the doc tag in figure 8.2 on page 54 it is noticable that not all of those putq's actually printed something. This is because The put statement always

pairs variables with the any string that immediately precedes it. If the variable does not have a value, then the string is not printed. This is another behavior of put that is perfect for printing HTML and XML attributes.

It's not so perfect behavior for debugging though. It is sometimes desirable to be sure that the put statement executed and that there was nothing to print. In that case, all that is needed is to separate the label and the variable by inserting a shorter label inbetween. This example separates the = from the label by making them into separate strings. The output will be the same, except 'Name' will always print even when the name variable has no value. Another technique is to separate the string from the variable by using two put statements.

```
put 'Name' '=' name;

put 'Name=';
put name;
```

8.2.2 The Default Event

The next event is the basic event. The basic event won't be found in any output from event_map. It's not one of the known events that ODS will request. But it is the event assigned to the default_event attribute. That means that the if the event is not initialize or Doc then the 'basic' event will handle the event request.

The basic event makes use of another variable 'empty'. If empty is true then this event does not encapsulate other events. In other words it's start and finish are triggered back to back with nothing in between. This knowledge is used to make the XML tag an empty[1] tag.

The value of empty is also used eliminate the ndent at the end of the start, and the entire finish section. The ndent statement tells the tagset to increment it's indention level by one. The amount of space indented depends on the value in the indent attribute at the top of the tagset. The xdent statement causes the indention level to decrease by one.

Finally, There is the last event attr_out. Attr_out is triggered from the doc and basic events and is responsible for printing the name and label attributes within each tag. This event is mostly for convenience. It makes it easy to add more attributes to the map just by inheriting from this tagset and redefining this Event.

Listing 8.4: Short Map Tagset

```
define tagset tagsets.short_map;
    notes "This is the event map definition";
    map = '<>&"';
    mapsub = '/&lt;/&gt;/&amp;/&quot;';
    nobreakspace = ' ';
    split = ' ';
    indent=2;
    stacked_columns = yes;
    output_type = 'xml';

    default_event = 'basic';

    define event initialize;
    end;
```

[1] This is what a XML tag that contains no other tags is called

```
    define event show_charset;
       set $show_charset "1" / if !exists(suppress_charset);
    end;

    define event doc;
        start:
            trigger show_charset;
            put '<?xml version="1.0"';
            putq " encoding=" encoding / if exists($show_charset);
            put "?>" CR CR;
            put "<" EVENT_NAME ;
            putq " title="          BODY_TITLE;
            putq " author="         AUTHOR;
            putq " operator="       OPERATOR;
            putq " sasversion="     SASVERSION;
            putq " saslongversion="    SASLONGVERSION;
            putq " date="           DATE;
            putq " time="           TIME;
            putq " encoding="       ENCODING / if exists(\$show_charset);
            putq " language="       LANGUAGE;
            putq " trantab="        TRANTAB;
            trigger attr_out;
            put ">" CR;
            ndent;
        finish:
            xdent;
            put "</" EVENT_NAME '>';
    end;

    define event basic;
        start:
            put "<" EVENT_NAME ;
            trigger attr_out;
            put "/" / if exists(empty);
            put ">" CR;
            break / if exists(empty);
            ndent;
        finish:
            break / if exists(empty);
            xdent;
            put "</" EVENT_NAME ">" CR;
    end;

   define event attr_out;
      putq " name=" name;
      putq " value=" value;
   end;
end;
```

8.2.3 summary

The mapping tagsets are the best resource for finding events are not known. They reveal all the events for any given SAS job. There are lots of ways to find events. One of the easiest ways is to examine the existing tagsets and their output. The mapping tagsets are another resource that can be extremely helpful. Adding a default event to an existing tagset can also be extremely helpful for revealing new events between the events already defined.

8.3 Finding Variables

In addition to knowing when events occur it's also important to know what data is available for any given event. The mapping tagsets show some of this but there are far too many variables to print them all the time.

There is one tagset that prints all the variables for all the events. The Odsxrpcs tagset uses a statement called putvars to do this.

To keep our example short this tagset defines only one event. The table event is familiar so that is a good place to start. The ODSxrpcs tagset uses a similar event for it's default event.

The entire tagset is not much more than a single putvars statement. See figure 8.5.

Listing 8.5: Printing Event Variables

```
proc template;

    define tagset tagsets.my_map;

        define event table;
            putvars event _name_ ' : ' _value_ nl;
        end;

     end;
run;

ods tagsets.my_map file='putvars.txt';
proc print data=sashelp.class; run;
ods tagsets.my_map close;
```

The output in figure 8.6 shows all the populated event variables for the print procedure's table event.

Listing 8.6: Output: Table Event Variables Output

```
type : table
anchor : IDX
event_name : table
encoding : iso-8859-1
operator : eric
date : 2003-10-25
sasversion : 9.1
saslongversion : 9.01.01M0D10062003
time : 01:00:07
state : start
proc_count : 1
```

```
total_proc_count : 1
page_count : 1
total_page_count : 1
dest_file : body
bodyname : putvars.txt
tagset : TAGSETS.MY_MAP
style : Default
javadate : 2003-10-25
javatime : 01:00:07-04:00
data_viewer : Report
style_element : Table
```

Adding the other variable category's and some indentation makes the output more informative and readable. The better version of the tagset. is shown in figure 8.7 on page 62. The new, nicely formatted, output is shown in figure 8.7 on page 62.

Listing 8.7: Printing all Variables

```
proc template;

    define tagset tagsets.my_map;

        indent=2;

        define event table;
            trigger put_all;
        end;

        define event put_all;
            put 'Event: ' event_name nl;
            ndent;

            put 'Event:' nl;
            ndent;
            putvars event _name_ ' : ' _value_ nl;
            xdent;

            put 'Style:' nl;
            ndent;
            putvars style _name_ ' : ' _value_ nl;
            xdent;

            put 'Memory:' nl;
            ndent;
            putvars memory _name_ ' : ' _value_ nl;
            xdent;

            put 'Stream:' nl;
            ndent;
            putvars stream _name_ ' : ' _value_ nl;
```

```
            xdent;

            put 'Dynamic:' nl;
            ndent;
            putvars dynamic _name_ ' : ' _value_ nl;
            xdent;

            xdent;
            put nl;
        end;

    end;
run;

ods tagsets.my_map file='putvars.txt';
proc print data=sashelp.class; run;
ods tagsets.my_map close;
```

Listing 8.8: Output: Table Event Variables Output

```
Event: table
  Event:
    type : table
    anchor : IDX
    event_name : table
    encoding : iso-8859-1
    operator : eric
    date : 2003-10-25
    sasversion : 9.1
    saslongversion : 9.01.01M0D10062003
    time : 01:13:14
    state : start
    proc_count : 1
    total_proc_count : 1
    page_count : 1
    total_page_count : 1
    dest_file : body
    bodyname : putvars.txt
    tagset : TAGSETS.MY_MAP
    style : Default
    javadate : 2003-10-25
    javatime : 01:13:14-04:00
    data_viewer : Report
    style_element : Table
  Style:
    FRAME : BOX
    RULES : GROUPS
    HTMLCLASS : Table
    CELLSPACING : 1
    CELLPADDING : 7
    FRAMEBORDER : auto
```

```
    CONTENTSCROLLBAR : auto
    BODYSCROLLBAR : auto
Memory:
Stream:
Dynamic:
```

8.4 The Putlog Statement

This chapter has shown how to reveal the inner workings of tagsets. There is another statement that is quite handy for quickly showing what is going on. Putlog is like put except that instead of printing to the output file it prints to the SAS log.

Putlog is nice for checking values or the order of events without putting debug messages in the output file. It also works nicely for printing usage messages from the tagset.

8.5 Define, Identify, Locate, Explore, and Solve.

This section will outline a method to solving problems with tagsets. The key to any solution is knowing what events are in the places that can help with the solution, and what information those events have.

1. **Define:**
   (a) **Understand the problem.** The first step is to understand what the desired output should look like. Modifying ODS output to the desired format is the easiest way to verify the goal.
2. **Identify:** Finding the events that need modification or definition is the first step towards solving any problem with a tagset.
   (a) Find the Markup output that needs changing.
   (b) Identify a unique string or data value
   (c) **Identify the event:**
   (d) **Search the Source.** If the string is not data driven, it is sometimes reasonable to find the string in the original tagset.
   (e) **Search the Mapping Output.** Run the SAS job using a mapping tagset and search for a known string. This is particularly useful if the SAS job provides a unique string to search for.
3. **Locate the event:**
   (a) locate the event definition in the source. Find the event that was identified in the mapping tagset's output in the parent tagset.
4. **Explore:**
   (a) **Copy the Parent's definition.** Copy and paste the parent tagset's event definition into the new tagset definition. If the identified event is not defined by the parent, then define it.

(b) **Identify Resources.** The next step is to figure out what to change in the event. Sometimes the changes are obvious. Regardless, it is always helpful to know what variables are bundled in the event request and what their values are.

i. **Identify Variables.** Adding a putvars statement or triggering a put_all type event is the next step. Run the SAS job using the new tagset, and look at the output file for the new event. Now there is a list of values that should enable a determination of the final event definition.

5. **Problems**

(a) **Use putlog.** If there are problems it is sometimes useful to add a putlog statement to see if things are going right. More putvars statements for more variable categories is also helpful.

(b) **Add a default event.** It is sometimes useful to add a default event that prints the event name. Any undefined events will then show up between the events already defined. If an event looks useful, add a definition with a putvars statement to see what sort of data the event contains.

6. **Solve:**

(a) **Final version.** Remove any debugging statements and make the final adjustments.

8.5.1 Repeat as Necessary

Most of the work happens in the identify, locate, and explore phase of the process. Usually some new tagset code results from knowledge provided by those three steps. But that is not usually the final solution. It is quite common to repeat these steps several times in the course of writing a new tagset.

8.6 Going step by step

Many of the problems that tagsets solve are fairly simple and involve a few events at most. This next example is representative of most common type of tagset modification. The key to solving this sort of problem is not so much the coding, but finding the place to put the code. The example will show how the steps of Define, Identify, Locate, explore and solve, are used.

8.6.1 Adding a Target to a URL

The Report procedure has a very handy option that allows dynamic creation of URL's based upon data. This feature works great for creating drill down reports that load successively detailed tables. But if the desire is for the page to load in a new window this approach will not work. This is because the target attribute will not carry through no matter how it is set. This can be fixed with a tagset.

8.6.2 Define the Problem

The following code shows Report procedure code that dynamically assigns a url based upon the data in each observation. Despite various efforts to give the url a target attribute of '_blank_' all efforts have failed. The result is that clicking on a link replaces the current page with the new one. The desire is to get another window with the new page. Looking forward to the next step, this job has already added an additional ods statement, event_map.

```
data x; x=1; x2=1; x3=1; x4=1; run;

ods html file="report_url.html";
ods tagsets.event_map file="report_url.xml";

proc report data=x nowd;
  column x x2 x3 x4;
  define x / display;
  define x2 / display;

  compute x2;
    call define( _COL_, "URL", "URL-from-REPORT" );
    endcomp;

  run;

ods _all_ close;
```

This is just a simple, do nothing, example. But it gives us an HTML table cell that looks like this.

```
<td class="r Data"><a href="URL-from-REPORT">          1</a>
```

There are various methods of setting the target style attribute on a column. However, none of them will work with a dynamic url. Our goal is a cell that looks like this.

```
<td class="r Data"><a href="URL-from-REPORT" target="_blank_">          1</a>
```

8.6.3 Identify the event

Looking in the html tagset source for 'href' reveals several events that could be the one. A little intuition might allow an accurate guess, but using the event_map tagset will remove all doubt. Adding the event_map ods statement to the job is all that is needed. Looking for - 'URL-from-REPORT' reveals these events.

```
<data event_name="data" trigger_name="attr_out" output_name="Report"
      output_label="Detailed and/or summarized report"
      section="body" class="Data" row="2" data_row="1"
      colcount="4" col="2" type="string" index="IDX" just="r">

  <hyperlink event_name="hyperlink" trigger_name="attr_out"
             output_name="Report"
             output_label="Detailed and/or summarized report"
             value="         1" colcount="4" index="IDX"
             just="c" url="URL-from-REPORT"/>
</data>
```

This is quite verbose but it tells exactly which event is needed, the hyperlink event. Finding events is an important skill when it comes to working with tagsets. This method is one of the primary ways to do that.

8.6.4 Locate the Event

Now that the event is identified the next step is to find it in the html tagset. Technically ODS HTML uses the html4 tagset, but that tagset inherits most of it's events from the htmlcss and phtml tagsets. Thankfully all

of those tagsets come bundled in one file, htmltags.tpl. This is another disadvantage of using the template procedure's source statement. It would take three source statements to see all of the events in the html4 tagset. The htmltags.tpl source file has everything in one place.

```
proc template;
source tagstes.html4;
source tagstes.htmlcss;
source tagstes.phtml;
run;
```

Searching the file finds three hyperlink events. One in phtml, a slightly more complex definition in htmlcss and a completely different definition in mvs_html. When working with the html4 or htmlcss tagsets, the event definitions are always more complex than their phtml counterparts. The event defined in htmlcss is the one that is needed since that is the one that html4 will use. It looks like this.

```
define event hyperlink;
  start:
    put '<a href="' URL;
    /* put "#" ANCHOR; */
    put '"';
    putq " target=" HREFTARGET;
    put ">";
    trigger do_value;
  finish:
    put "</a>" CR;
end;
```

8.6.5 Explore the Data

Adding a putvars statement or triggering put_all doesn't really help much. What it shows is that even if hreftarget is set in the style, The report procedure removes it.

The target line in the hyperlink event needs to have an alternate behavior. Causing the alternate behavior by indicating a table column will do. Macro variables are the easiest way to do that. One macro variable can tell which column needs the target, and another can give the target value.

```
%let target_value=_blank_;
```

Adding this to the hyperlink event should get us close. But it will have to be conditional. So this isn't a complete answer.

```
putq " target=" target_value;
```

8.6.6 Repeat. Identify, Locate, Explore

The hard part is over. The event has been identified, located and the data it contains is known. However, this is only a partial solution. The final solution needs to preserve the current behavior while adding a new feature that is driven by macro varibles.

But there is still one problem. The hyperlink event doesn't know anything about table columns. That will make it difficult to cause the alternate behavior.

Looking at the original event_map output it can be seen that the data event knows the column number. Colstart was used earlier in the CSV examples. It can be used again here. But it will have to be saved away in a variable. The next step is to copy and paste the data event from the htmlcss tagset. The only changes are to set $column to colstart and to unset it at the end. Now the hyperlink event can use the $column variable to check the column number against the new macro variable.

8.6.7 The solution

Looking at the new hyperlink event and the new target_column and target_value variables it should be fairly obvious that these variables tell which column to add a target to, and what that target should be.

```
define tagset tagsets.targethtml;
    parent=tagsets.html4;

    mvar target_column;
    mvar target_value;

    define event hyperlink;
        start:
            put '<a href="' URL;
            /*put "#" ANCHOR;*/
            put '"';

            do /if hreftarget;
                putq " target=" HREFTARGET;
            else /if cmp($column, target_column);
                putq " target=" target_value /if target_value;
            done;

            put ">";
            trigger do_value;
        finish:
            put "</a>" CR;
    end;
end;
```

This tagset will work perfectly most of the time. The exception is the report procedure. The report procedure frequently gives a data event with no data. Then does something else between the start and finish of the data event. The only real problem with that is the metadata and the context that metadata provides. In this case column number of the cell was needed. The set statement is used to set $column to colstart. When the hyperlink event comes along, it has something to check against the macro variable. The good news is that the data event, with the exception of the set $column statement, is an exact copy of the data event from the htmlcss tagset.

Running the following proc print with the event_map tagset will shows a different data event than the one provided by report. Ultimately the data event triggers the hyperlink event, so our solution still works. But for print, it would have been unessecary to save away the column number.

```
proc print data=x;
    var x;
    var x2 / style(data) = [url="URL-from-Print"];
```

```
    var x3;
    var x4;
run;
```

The XML created for a data cell looks like this.

```
<data event_name="data" trigger_name="attr_out" output_name="Print"
  output_label="Data Set WORK.X" section="body" class="Data" value="1"
  row="2" data_row="1" colcount="1" col="3" type="double"
  rawvalue="P/AAAAAAAAA=" index="IDX1" just="r" url="URL-from-Print">
</data>
```

Realizing that this needed to be done for the report procedure only comes with experience. This problem will show up several times in this book. At some point, realizing that the report procedure is special will become second nature. This problem will go away in the future, as improvements are made to proc report. The final solution is shown below.

Listing 8.9: The complete Target HTML tagset

```
proc template;
    define tagset tagsets.targethtml;
        parent=tagsets.html4;

        mvar target_column;
        mvar target_value;

        define event hyperlink;
            start:
                put '<a href="' URL;
                /* put "#" ANCHOR; */
                put '"';

                do /if hreftarget;
                    putq " target=" HREFTARGET;
                else /if cmp($column, target_column);
                    putq " target=" target_value /if target_value;
                done;

                put ">";
                trigger do_value;
            finish:
                put "</a>" CR;
        end;

        define event data;
            start:
                trigger header /breakif cmp(htmlclass, "RowHeader");
                trigger header /breakif cmp(htmlclass, "Header");

                /* The only new line here */
                set $column colstart;
```

```
                    put "<td";
                    putq " title=" flyover;
                    do /if !cmp(htmlclass,'batch');
                        trigger classalign;
                        trigger style_inline;
                    done;
                    trigger rowcol;
                    put " nowrap" /if no_wrap;
                    put ">";
                    trigger cell_value;
                finish:
                    trigger header /breakif cmp(htmlclass, "RowHeader");
                    trigger header /breakif cmp(htmlclass, "Header");

                    trigger cell_value;
                    put "</td>" CR;

                    /* The only new line here - unset just to be safe */
                    unset $column;

                end;

    end;
run;


data x; x=1; x2=1; x3=1; x4=1; run;

%let target_column=2;
%let target_value=_blank_;

ods tagsets.targethtml file="report_url.html";

proc report data=x nowd;
  column x x2 x3 x4;
  define x / display;
  define x2 / display;

  compute x2;
    call define( _COL_, "URL", "URL-from-REPORT" );
    endcomp;

  run;

ods _all_ close;
```

The output gives the desired reselt. The same cell in the output now looks like this.

```
<td class="r Data"><a href="URL-from-REPORT" target=_blank_>          1</a>
```

Notice that putq didn't do it's job. This appears to be a bug with macro variables in SAS 9.1. The best solution for that is to put the quotes around the variable yourself. Using a put instead of a putq will insure

that the tagset won't break when a future release fixes the quoting bug.

```
do /if hreftarget;
    putq " target=" HREFTARGET;
else /if cmp(\$column, target_column);
    put ' target="' target_value '"' /if target_value;
done;
```

Another method is to assign the macro variable to another variable with a set. Then use that variable in the putq.

```
do /if hreftarget;
    putq " target=" HREFTARGET;
else /if cmp($column, target_column);
    do /if target_value;
        set $target_value target_value;
        putq ' target=' $target_value;
        unset $target_value;
    done;
done;
```

8.7 Summary

Tagsets provide some powerful tools which can help find the best way to write or change any tagset. When starting from scratch one of the mapping tagsets is a good place to start. They are also good for comparing against existing tagsets.

The default event can be used in an existing tagset to reveal unknown events between the events already defined.

The putvars statement can be used to show all the currently populated variables. Including a put_all event in a tagset gives a quick and easy way to find that elusive variable when needed.

The putlog statement can be used to print information to the log which can make it easier to find or see variables or sequences of events. They are particularly useful as a sanity check, when it's not clear that a tagset is doing what it should.

Chapter 9

File Redirection

Adding output to the all the files specified on the ODS statement requires that some events be redirected to those files. This chapter will show the differences between the various files and the events associated with them. It will also show how to use file redirection to modify their contents.

Most event requests are sent to the body file specified on the ods statement. The other files only get a skeletal number of events. The stylesheet file is an exception to that. A complete stylesheet file can be created with an ods statement that looks like this.

```
ods html stylesheet="style.css";
```

This statement gives an error, but the stlyesheet is still created. There is one tagset that was meant to be used this way. The odsstyle tagset only defines stylesheet events. Even if a body file is specified, nothing will be written to it.

Creating a table of contents is one of the more common desires, yet the contents file receives none of the event requests needed to create one. This is great for document types like latex and XML that would rather have the hierarchical information embedded in the body of the file. It is also not difficult to create a separate file if desired.

9.1 The File Attribute

Each event can have a file attribute that redirects which file the event's output will go to. The attribute values correspond to the names given on the ods statement.

Value	File / Action
Contents	The contents file
Pages	The pages file
Frame	The frame file
Code	The Code file
Data	The Data file
Stylesheet	The Stylesheet file
Body	The body file

Once an event has been redirected all events triggered will also be redirected until the redirected event is done.

9.2 Identify

Running the short_map tagset will show that there are several events that give structure to the output. The structure is even easier to see if the table events are eliminated. The new tagset looks like this. It might have been easier to define the desired events instead of using the default. But this way, it's a sure bet nothing important has been eliminated. For variety, this job uses the GLM procedure. This procedure creates a fairly complex table of contents. The ODS select statement has been used to shorten the output considerably.

```
proc template;
    define tagset tagsets.leaf;
       parent = tagsets.short_map;
       image_formats = "gif,jpeg";

       define event data; end;
       define event row; end;
       define event table_body; end;
       define event table_head; end;
       define event cell_is_empty; end;
       define event rowspec; end;
       define event colgroup; end;
       define event colspecs; end;
       define event colspec_entry; end;
       define event colspecsep; end;
       define event header_spec; end;
       define event span_header_colspec; end;
       define event span_group; end;
       define event row_group; end;
       define event sub_rowheader_colspec; end;
       define event row_header_spec; end;
       define event col_header_label; end;
       define event cellspec; end;
       define event cellspecsep; end;
       define event doc_head; end;
       define event doc_meta; end;
       define event auth_oper; end;
       define event doc_title; end;
       define event stylesheet_link; end;
       define event code_link; end;
       define event javascript; end;
       define event startup_function; end;
       define event shutdown_function; end;
       define event table_headers; end;
       define event col_header_spec; end;
       define event sub_header_colspec; end;
       define event sub_colspec_header; end;
       define event header; end;
       define event cellspecspan; end;
       define event colspanfillsep; end;
       define event colspanfill; end;
       define event rowspancolspanfill; end;
```

```
        define event rowspanfill; end;
        define event system_title_setup_group; end;
        define event system_title_setup; end;
        define event system_footer_setup_group; end;
        define event system_footer_setup; end;
        define event title_setup_format_section; end;
        define event title_format_section; end;
        define event page_setup; end;
        define event title_setup_container; end;
        define event title_setup_container_specs; end;
        define event title_setup_container_spec; end;
        define event title_setup_container_row; end;
        define event title_container_row; end;
        define event title_container_specs; end;
        define event title_container_spec; end;
        define event put_value; end;
    end;
run;

    data plants;
       input type $ @;
       do block=1 to 3;
          input stemleng @;
          output;
          end;
       cards;
    clarion   32.7 32.3 31.5
    clinton   32.1 29.7 29.1
    knox      35.7 35.9 33.1
    oneill    36.0 34.2 31.2
    compost   31.8 28.0 29.2
    wabash    38.2 37.8 31.9
    webster   32.5 31.1 29.7
    ;

    data mileage;
       input mph mpg @@;
       cards;
    20 15.4 30 20.2 40 25.7 50 26.2 50 26.6 50 27.4 55  . 60 24.8
    ;

options ls=100;

ods select  GLM.Data.ClassLevels;
ods select  GLM.Means.type.stemleng.MCLines.Waller.MCLinesInfo;
ods select  GLM.Means.type.stemleng.MCLines.Waller.MCLines;


title 'A system title';
title2 'Another system title';
```

```
footnote 'A system Footer';
footnote2 'Another system Footer';

ods tagsets.event_map file="outline2.xml";
ods tagsets.leaf file="outline.xml";

   proc glm order=data data = plants;
      class type block;
      model stemleng=type block / solution;
      means type / waller regwq;

   *-type-order-------------clrn-cltn-knox-onel-cpst-wbsh-wstr;
      contrast 'compost vs others' type -1 -1 -1 -1  6 -1 -1;
      contrast 'river soils vs.non' type -1 -1 -1 -1  0  5 -1,
                                    type -1  4 -1 -1  0  0 -1;
      contrast 'glacial vs drift'   type -1  0  1  1  0  0 -1;
      contrast 'clarion vs webster' type -1  0  0  0  0  0  1;
      contrast 'knox vs oneill'     type  0  0  1 -1  0  0  0;
   run;

ods _all_ close;
```

The output from this tagset is very interesting. The most obvious events are the proc, proc_branch, branch and leaf events. It is those last three that are used by the html tagsets to create the table of contents.

```
<?xml version="1.0" encoding="iso-8859-1"?>

<doc operator="eric" sasversion="9.1"
     saslongversion="9.01.01M0D10062003"
     date="2003-11-17" time="01:59:01" encoding="iso-8859-1">
  <doc_body>
    <proc name="GLM">
      <anchor name="IDX"/>
      <system_title_group>
        <title_container>
          <system_title value="A system title">
          </system_title>
          <system_title value="Another system title">
          </system_title>
        </title_container>
      </system_title_group>
      <proc_title_group>
        <proc_title value="The GLM Procedure"/>
      </proc_title_group>
      <proc_branch name="GLM" value="GLM">
        <branch name="Data" value="Data">
          <leaf name="ClassLevels" value="Class Levels">
            <page_anchor/>
            <output name="ClassLevels">
              <table>
              </table>
```

```
    </output>
  </leaf>
</branch>
<system_footer_group>
  <title_container>
    <system_footer value="A system Footer">
    </system_footer>
    <system_footer value="Another system Footer">
    </system_footer>
  </title_container>
</system_footer_group>
<pagebreak/>
<anchor name="IDX1"/>
<page_anchor/>
<system_title_group>
  <title_container>
    <system_title value="A system title">
    </system_title>
    <system_title value="Another system title">
    </system_title>
  </title_container>
</system_title_group>
<proc_title_group>
  <proc_title value="The GLM Procedure"/>
  <proc_title/>
  <proc_title value="Waller-Duncan K-ratio t Test for stemleng"/>
</proc_title_group>
<branch name="Means" value="Means">
  <branch name="type" value="type">
    <branch name="stemleng" value="stemleng">
      <branch name="MCLines" value="Multiple Comparison Lines">
        <branch name="Waller" value="Waller-Duncan">
          <leaf name="MCLinesInfo" value="Information">
            <output name="MCLinesInfo">
              <table>
              </table>
            </output>
          </leaf>
          <anchor name="IDX2"/>
          <leaf name="MCLines" value="Comparisons">
            <output name="MCLines">
              <table>
              </table>
            </output>
          </leaf>
        </branch>
      </branch>
    </branch>
  </branch>
</branch>
```

```
            <system_footer_group>
              <title_container>
                <system_footer value="A system Footer">
                </system_footer>
                <system_footer value="Another system Footer">
                </system_footer>
              </title_container>
            </system_footer_group>
          </proc_branch>
        </proc>
      </doc_body>
    </doc>
```

The events that can be seen in this map are the basis for all document structure. Redirecting these events or triggering other events that are redirected allows great flexibility in the creation of a table of contents or document outlines. Using event_map as the parent to this tagset would reveal more variables and values which could also be useful.

Among the variables that could be seen are toclevel, and url. Toclevel is a counter that indicates the contents depth. Url is the url for the current output object.

9.3 Locate and Explore

The next step is to look in the tagset source to see what these events are used for. The HTML tagsets do very little in the proc_branch, branch and leaf events. These events check the 'hidden' variable and trigger other events that are redirected to the contents file. An obvious and easy modification would be to block the branch event. The result would be a very flat table of contents. Depending on the Procedure, this is sometimes desirable.

```
proc template;
    define tagset tagsets.flat_contents;
        parent=tagsets.html4;

        define event branch;
        end;
    end;
run;
```

Another variation would be to block the branch entries based on the toclevel variable. It is possible to eliminate any levels desired. This example eliminates contents levels deeper than 3.

```
proc template;
    define tagset tagsets.flat_contents;
        parent=tagsets.html4;

        define event branch;
            start:
                break / if hidden;
                break / if inputn(toclevel, '7.') > 3;
                trigger contents_branch;
            finish:
```

```
                break / if hidden;
                break / if inputn(toclevel, '7.') > 3;
                trigger contents_branch;
            end;
        end;
    run;
```

Another interesting event is the page_anchor event. This event is redirected to the pages file to create links to each page of output. It might be desirable to redirect the page_anchor event to the contents file. This tagset will do that. This tagset preserves the original behavior by creating a new event from the original page_anchor event. Page_anchor_entry is exactly the same but without the file redirection. There are two new events whose sole purpose is to redirect the output from the new event to the appropriate file. The end result of this tagset is that page numbers are inserted as links in the table of contents.

```
proc template;
    define tagset tagsets.flat_contents;
        parent=tagsets.html4;

        define event page_anchor;
            start:
                trigger contents_page_anchor;
                trigger pages_page_anchor;
            finish:
                trigger contents_page_anchor;
                trigger pages_page_anchor;
        end;

        define event contents_page_anchor;
            file=contents;
            start:
                trigger page_anchor_entry;
            finish:
                trigger page_anchor_entry;
        end;

        define event pages_page_anchor;
            file=pages;
            start:
                trigger page_anchor_entry;
            finish:
                trigger page_anchor_entry;
        end;

        define event page_anchor_entry;
            start:
                put "<li";
                putq " class=" HTMLCLASS;
                put ">" nl;
                put '<a href="' ;
                put path_name /if !exists(path_url);
                put path_url;
```

```
                put body_name /if !exists(body_url);
                put body_url;
                put "#" ANCHOR;
                put '"';
                putq ' target="body" ';
                put ">";
                trigger pre_post;
                put 'Page ' total_page_count;
                trigger pre_post finish;
                put "</a>" CR;
            finish:
                put "</li>" nl;
        end;
    end;
run;
```

9.4 File interactions

The ODS markup statement allows for seven separate file specifications. Each of these files has certain interactions with the others. Specifying a stylesheet or code file will result in stylesheet_link and code_link event requests being sent to the body, contents, and pages files. A frame file will receive event requests providing information about the body, contents, and pages files if they were specified. The easiest way to see all of the file interactions is to run a job like this one. The event_map tagset may also be helpful.

Listing 9.1: Getting simple maps for all files

```
ods tagsets.short_map file='body.xml'
                      contents='contents.xml'
                          page='pages.xml'
                         frame='frame.xml'
                    stylesheet='style.xml'
                          code='code.xml'
                          data='data.xml';
ods tagsets.short_map close;
```

Looking at the maps created for each file will reveal the various places each file is referenced by another.

1. The stylesheet is referenced to by every file except the frame file.
2. The code file is referenced to by the body, contents, and pages files.
3. The frame file includes frame events for the body, contents and pages file.

9.5 Freedom of Choice

While there are events that are typically used to reference the other files, it is not necessary to use them to do so. All of the filenames and their url's are available at all times. The following event finish definition is from the html tagsets. It uses the file names and urls to create a noframe section for 508 compliance.

```
finish:
  put "</frameset>" nl;
  put "<noframes>" nl;
  put "<ul>" nl ;

  do /if any(contents_name,contents_url);
       put '<li><a href="';
       put basename "'";
       put path_url;
       put path_name /if ^path_url;
       put "(";
       put contents_url;
       put contents_name /if ^contents_url;
       put ")'" '"';
       put ">The Table of Contents</a></li>" nl;
  done;

  unset $link;
  do /if any(pages_name,pages_url);
       put '<li><a href="';
       put basename "'";
       put path_url;
       put path_name /if ^path_url;
       put "(";
       put pages_url;
       put pages_name /if ^pages_url;
       put ")'" '"';
       put ">The Table of Pages</a></li>" nl;
  done;

  put '<li><a href="';
  put basename "'";
  put path_url;
  put path_name /if ^path_url;
  put "(";
  put body_url;
  put body_name /if ^body_url;
  put ")'" '"';
  put ">The Contents</a></li>" nl;

  put "</ul>" nl ;
  put "</noframes>" nl;

  unset $link;
end;
```

9.5.1 Explore

Like most other things, A tagset and ods statement can be written that explores these variables. The following job will show the file variables that are available.

Listing 9.2: Exploring Filename Variables

```
proc template;
    define tagset tagsets.filenames;
        define event initialize;
            putvars event _name_ ': ' _value_ nl;
        end;
    end;
run;

ods tagsets.filenames file='body.xml' (url='b.html')
                    contents='contents.xml' (url='c.html')
                        page='pages.xml' (url='p.html')
                       frame='frame.xml' (url='f.html')
                  stylesheet='style.xml' (url='s.html')
                        code='code.xml' (url='cd.html')
                        data='data.xml' (url='d.html');
ods tagsets.filenames close;
```

Because the initialize event only happens once, it happens in the first file that is opened. That happens to be the code file. All the other files created are empty. But the output is quite interesting. Most of the variables are populated at all times.

```
event_name: initialize
empty: 1
encoding: iso-8859-1
operator: eric
date: 2003-11-17
sasversion: 9.1
saslongversion: 9.01.01M0D10062003
time: 21:28:53
state: start
dest_file: code
bodyname: body.xml
bodyurl: b.html
contentsname: contents.xml
contentsurl: c.html
pagesname: pages.xml
pagesurl: p.html
stylesheet: style.xml
stylesheet_url: s.html
codename: code.xml
codeurl: cd.html
dataname: data.xml
dataurl: d.html
framename: frame.xml
frameurl: f.html
stylesheet_title: style.xml
tagset: TAGSETS.FILENAMES
style: Default
javadate: 2003-11-17
javatime: 21:28:53-05:00
```

9.6 Summary

File redirection is an important feature of tagsets. Without it there would be no separate table of contents. But redirection can be used for more than just contents. The data and code files were created specifically to allow for separation of data, code and formatting if that is desired.

Part III

Technicalities

Chapter 10

The Tagset Attributes

A tagset can have several attributes. The attributes control the basic behavior of the tagset. Most importantly the translation of special characters that may interfere with the markup language being generated. Other attributes control output indention, breaking of lines, even the way the output should be displayed in the SAS output window. Setting up a tagset's attributes is the first step towards creating a new tagset.

10.1 Parent

The Parent attribute designates another tagset to inherit attributes and events from.

10.2 Special Characters

Every markup language has characters that it thinks are special. Usually these characters need to be remapped to something else. The map attribute tells us which characters we care about. For HTML and XML the most important characters are <> and &. The mapsub attribute is a list of strings which correspond to the characters in the map attribute. Each string is delimited by a character of your choosing. The only requirement is that the value of mapsub start with that character.

```
map='\&<>';
mapsub='/\&amp;/\&lt;/\&gt;/';
```

10.2.1 Automatic Character Translation

The old ODS HTML destination only translated special characters if the string it had did not start with < and end with >. This feature of HTML is set in the style templates, and is set using the protectspecialchars attribute. The valid values for protectspecialchars is yes, no, and auto. The auto setting is when this behavior comes into play.

This automatic non-translation allowed for HTML code to be put directly into the dataset, or style. When ODS HTML detected raw HTML code it would leave it alone. This same mechanism needs to work for ODS Markup. But what determines well formed raw markup changes with the markup being generated. To support this feature there are two attributes that can be set in the tagset. beginWellFormed

and endWellFormed. BeginWellFormed is what the string should start with, and endWellFormed is what the string should end with. If the string matches, ignoring leading and trailing whitespace, then the characters in the string will not be translated. If these attributes are not set, then the result is that strings are always translated.

```
BeginWellFormed='<';
EndWellFormed='>';
```

10.3 Non Breaking Spaces

Many markup languages have something called a non-breaking space. In HTML a non-breaking space is . In latex it is ~ The nobreakspace attribute is how you tell the tagset what it should use if need be. If nobreakspace is not set then the normal space character is used.

```
nobreakspace=' ';
```

10.4 Split Characters

Sometimes SAS wants to break strings into multiple lines. The split attribute tells us what to do when that happens. In HTML the value of split is '
' in latex it's
linebreak. For many output types a space is appropriate.

```
split='<br>';
```

10.5 Indention

The indention attribute is purely to make the markup output pretty. This attribute determines how many spaces to use when increasing or decreasing the indent level of the output, by using the Ndent and Xdent commands.

```
indent=4;
```

10.6 Stacked Columns

Stacked columns are a concept used by table templates that allow multiple columns of a table to be combined into one column with the values stacked on top of each other. Some forms of markup do not allow for this sort of thing. The stacked_columns attribute allows the tagset to turn stacked columns off if they would cause problems with the output. The result is that the columns will be printed separately. The default is for stacked columns to be on. if on the values tend to be printed within the data event with the value of the split attribute inbetween them.

One caveat is that no matter what the setting, the Freq procedure will still do stacked columns.

```
stacked_columns=no;
```

10.7 Image Formats

The value of image_formats is to determine which types of images the destination will support. The image formats are listed in order of preference. When graphics are being generated for the destination, the preferred type is checked against this list. If the type is in the list everything goes as planned. If the type is not in the list then the first type in the list will be used instead. This is particularly useful for the LaTeX and troff destinations. Those destinations do not always support the more generally acceptable gif and jpeg formats.

```
image_formats='gif,jpg,png';
```

10.8 Output Type

The value of Output_type is for the SAS display manager. It gives SAS clues about how to treat the output files the tagset creates. Known values for output type are HTML, XML, Latex, troff, and CSV.

10.9 Adding Measurements

The measurement attribute indicates whether measurements of the output should be supplied. This attribute is currently ignored.

10.10 Copyright Symbol

The Copyright attribute should be set to a string that will render to a copyright symbol for the markup being generated.

10.11 Trademark Symbol

The trademark attribute should be set to a string that will render to a trademark symbol for the markup being generated.

10.12 Registered Trademark Symbol

The Registered_Trademark attribute should be set to a string that will render to a registered trademark symbol for the markup being generated.

10.13 Default Event

The default event is the name of an event to use if the event requested by ODS is not defined. Some tagsets consist almost entirely of just a default event.

10.14 Embedded Stylesheet

This attribute tells ODS that it is ok to trigger stylesheet events even when there is no stylesheet file. If Embedded_stylesheet is set to yes, and there is no stylesheet specified on the ODS statement, then stylesheet events will be directed to the body file.

10.15 Pure Style

This attribute causes the default behavior of ODS Markup to be like the old ODS HTML. A stylesheet is not used. Although it could be generated. The result is that all style attributes are available on all events. Normally, the styles definitions are printed at the beginning. Later on most of the style attributes will be blanked when the style is actually in use. For instance the font and foreground color will not be defined when the titles actually occur. Only the name of the style to be used will be populated. Any style over rides will be seen. But for the most part that is all that will show up other than the actual style name.

The best way to visualize this is to create an HTML tagset that does it. Extremely verbose HTML is the result. This is not a good way to create HTML but there may be markup languages or other uses that require this behavior. Although, I have yet to see an instance where this behavior is actually desired.

```
proc template;
    define tagset tagsets.verbose_html;
        parent=tagsets.html4;
        embedded_stylesheet=no;
        pure_style=yes;
    end;
run;
ods html file='simple.html';
ods tagsets.verbose_html file='verbose.html';
ods _all_ close;
```

The result is extremely verbose output. The body tag from the purestyle tagset is shown first, followed by the same tag from the html4 tagset.

```
<body onload="startup()" onunload="shutdown()"
      style=" font-family: Arial, Helvetica, sans-serif; font-size: 3;
      font-weight: normal; font-style: roman; color: #002288;
      background-color: #E0E0E0; margin-left: 8; margin-right: 8;">


<body onload="startup()" onunload="shutdown()" class="Body">
```

10.16 Package

The package attribute sets a package template to use with this destination. When this happens, a package that uses that package template will be created. If the package is an archive then it will be given the file name from the ODS statement. The actual files created and placed in the package will take their names from the tagset's file name attributes. By default, the package created will be an archive.

10.17 File Names

There are 7 file names that can be specified within the tagset. One for each of the files that can be created by ODS Markup. Body, contents, pages, frame, stylesheet, code and data. The filenames are specified as quoted strings.

10.18 Mime Types

The mime types for each of the 7 files can also be specified. There is also a default mime type that will be used in the absence of the others. If the mime types are not specified ODS will guess at the mimetype for the file. These attributes are; body_mimetype, contents_mimetype, pages_mimetype, frame_mimetype, stylesheet_mimetype, code_mimetype, data_mimetype, and default_mimetype.

10.19 Measurement"

Setting measurement to yes will turn on the measurement of the content being generated. All text will be measured for height and width and will be adjusted to fit the physical page. Tables will be paneled if needed. This behavior is what has made it possible to create an RTF tagset. The MLaTeX tagset also takes advantage of measurement to create reports that fit nicely on paper.

10.20 Log_note

The lognote attribute is a note that will be printed once when the tagset is used in an ODS statement. The same behavior can be accomplished using the putlog statement in the initialize event.

10.21 Splitting Text

ODS HTML introduced a new behavior with SAS 8.1. Text was automatically broken to the next line if the length exceeded the width of the column or page. This behavior worked well most of the time. But not always. Tagsets give complete control over this behavior with the three breaktext attributes. This is a fairly complex set of attributes. The first two are to disable the breaking of text based on the length of the string or the width of the space the string is going into. The third defines when text should be broken based on the ratio of space verses text. The best way to see how they work is to play with them.

10.21.1 Breaktext Length

Breaktext_length is the maximum **length of text** which will be considered for placement of automatic breaks. If the text is longer than this value then no breaks will be inserted automatically To keep text longer than 66 characters long from having forced breaks inserted, you would specify:

```
breaktext\_length=66;
```

10.21.2 Breaktext Width

Breaktext_width is the maximum **width of space** that will be considered for placement of automatic breaks in text. If the width of the space is greater than this value the text will not be broken. To not break text that is going into a space greater than or equal to 40 characters wide, you would specify:

```
BreakText_Width=40;
```

10.21.3 Breaktext Ratio

Breaktext_Ratio is the ratio of the width of space to the length of the text which is supposed to fit in it. Like the other two attributes, this attribute serves to narrow the the string and width combinations that will be considered for splitting. The text length and width must fall within this ratio before they will be considered for forced splits.

If it is desirable to never force breaks in text which is less than 1.2 times longer than the width of space it is to fit in, you would specify:

```
BreakText_Ratio=1.2;
```

10.22 External Graph Instance

The External_graph_instance attribute is used specifically to save on memory usage. Setting this attribute to yes, means that when graph procedures are used, another ods Markup file will be created internally. The graph procedure will use the graph tagset to generate it's output, and that output will be incorporated into the output generated on the ods statement. This comes into play when the devices are set to java, javaimg, activex and activeximg. It is also used by statistical graphics. If a tagset does not specify this, then only images will be supported by the tagset, unless it defines the graph events.

10.23 No Byte Order Mark

When creating XML output with a unicode encoding a byte order mark should be the first character in the file. The byte order mark tells the XML parser what the byte order is for multibyte characters. When viewed in an editor it looks like hexadecimal characters. Many inferior XML parsers cannot cope with the byte order mark. If this attribute is set to yes, the generated XML will not have a byte order mark. This is a bad idea. The pain and suffering caused by actually typing in and reading the attribute says it all. 'No_Byte_Order_Mark=yes'.

10.24 Hierarchical Data

Setting the Hierarchical_Data attribute to yes will cause the Tabulate and Freq procedures to follow a hierarchical event model instead of a tabular event model. This model allows the creation of data cubes ala OLAP.

10.25 Summary

There are many attributes available but for the most part they can be set and forgotten or even better, inherited from another tagset.

Chapter 11

Creating Variables

The set statement has already been used in several examples. Set is the primary way to create and modify variables in tagsets. Before continuing it would be good to explain the different types of variables and the methods of creating them. This chapter is dedicated to explaining the different types of variables that can be created and used in tagsets.

The first step towards understanding is to create a tagset that we can play with. We know the initialize event happens once and that it happens at the very beginning. We can create a framework that will allow us to play with tagset just by utilizing the the initialize event. There are other events we could use. Doc for instance. But initialize will do just fine. All of the code snippets can be placed directly into the initialize event in the following code.

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

          define event initialize;

               /* put play code here */

          end;
     end;
run;

ods tagsets.variables file='test.txt';
ods tagsets.variables close;
```

11.1 String Variables

The most common variables are string variables. Any variable created with the set statement will be a string. Tagset variables are variable length, meaning that they are always the size of the text they contain. If a variable is assigned an empty string then it really doesn't exist at all. But that's ok, because referencing a non-existant variable is allowed, in fact it's the easiest way to create conditional values. Setting a variable to 'true' and unsetting it for false is the most efficient way to use a variable of this sort.

The set statement always creates strings. The first argument is the variable to set. All variables that can be created are prefixed with a $ or a $$. After that, the set statement works just like a put, any number of strings and variables can be specified. Including the variable that is being assigned.

The unset statement is used to delete any variables that may have created. Unsetting a variable that does not exist does no harm.

```
set $test 'The value is:' value;
set $test2 $test ' and the label is:' label;

unset $test;

put $test2;
```

Like the put statement, set can also have datastep functions. The functions cannot be nested.

```
set $test  operator ' ran this job.';
set $test2 $test ' On this date: ' date;
set $test3 compress($test2, ':.');
```

11.2 Lists

Set can also create variables that are lists. Lists are denoted by a suffix of '[]'. If the brackets are empty the set adds a new entry to the end of the list. Lists are 1 based, the first entry always has an index value of 1. If there is a number within the brackets the value at that location in the list will be replaced with the new value. If the list is not that long, then the value is appended to the end of the list. If a list variable is referenced without a subscript the return value is the number of entries in the list.

```
set $test[]  operator ' ran this job.';
set $test[] $test[1] ' On this date: ' date;
set $test[] compress($test[2], ':.');
```

Lists also accept negative index values. If the index value is negative the the list will be indexed from the end instead of the beginning. An index of -1 is always the last entry in the list. Unsetting a value in a list causes the list to shrink and all entries from that point on will appear to have moved to eliminate the gap. The last entry on a list can be deleted by unsetting the -1 entry.

```
set $test[] 'Entry 1';
set $test[] 'Entry 2';
set $test[] 'Entry 3';

set $test[2] 'The New 2';

put 'Test has a length of ' $test nl;

/* print them in reverse order */
put $test[-1] nl;
put $test[-2] nl;
put $test[-3] nl;

/* delete the first entry */
```

```
unset $test[1];

put 'Test has a length of ' $test nl;

/* delete the whole thing */

unset $test;

put 'Test has a length of ' $test nl;
```

11.3 Dictionaries

Dictionaries are just like lists except that their indexes are strings instead of numbers. These are called key's. In order to create an entry in a dictionary a key must be provided.

```
 set $test['one'] operator ' ran this job.';
 set $test['one plus'] $test['one'] ' On this date: ' date;
 set $test['one sentence'] compress($test['one plus'], ':.') '.';

 put 'Test has a length of ' $test nl;

 set $key 'one sentence';
 put $test[$key] nl;

unset $test;
```

11.4 Numeric Variables

Numeric variables can be quite useful for list manipulation and many other things. Numeric variables can be created with the eval statement. The eval statement has two arguments. The first is the variable to set. The second is a where clause. Eval is one of the few statements that cannot have an if condition. The problem arises because / is a valid character in a where clause. The do statement allows a solution to this problem. The type of the variable created with the eval statement depends on the where clause. If the result of the clause is numeric then the variable will be numeric, otherwise it will be a string. One advantage of where clauses is that functions can be nested. But a set statement is much more efficient, so if a string is desired set is the better choice.

```
set $key 'one sentence';
eval $count 1;
eval $count $count+1;
eval $position index($test, 's');
eval $string1 substr($key, $position);
set $string2 substr($key, $position);
```

11.5 Stream Variables

Occasionally the target output does not fit the event model used by tagsets. This is where stream variables are most useful. Stream variables are more like a temporary file than a variable. The set and unset statements work with stream variables, but it is much more common to create or append to a stream with the open statement. Once a stream is open all output will be directed to the stream. Streams automatically compress anything that is written to them. As long as the output is small the stream will stay in memory. As it grows it will be written to disk. A stream is closed with a close statement. A stream will be appended to if it is reopened. Only one stream can be open at a time, so opening a stream will close any stream that was previously open. When using a stream variable in a put or set or unset the variable is prefixed with a $$.

```
open deferred;
    put 'we will print this later.  Goodbye.' nl;
close;
put 'This will print now.' nl;

put $$deferred;

unset $$deferred;
```

11.5.1 Stream Specific Statements

There are several stream specific statements besides open and close. Stream specific commands don't use the $$ prefix because they know that they are operating on a stream. Delstream deletes a stream but the command is redundant because unset does the same thing. Putstream on the other hand can be useful. Putstream writes the given stream directly to the output file. If another stream is currently open, putstream will reopen that stream when it is done.

Another command that is sometimes needed is flush. Flush forces any buffered output to be written to the current output stream. This applies to the 'normal' stream that points to the output file, or to any other stream which may be open. This is sometimes necessary when switching between streams.

With the release of SAS 9.2 the open statement will also be able to accept variables as a stream name. This is similar to the concept of pointers in the C programming language. When a variable is given to an open statement a stream is opened with the name contained within the variable. Using list and dictionary variables with the open statement allows for easy management of multiple streams within a tagset. This simple example behaves just like the previous example.

```
set $mystream "deferred";

open $mystream;
    put 'we will print this later.  Goodbye.' nl;
close;
put 'This will print now.' nl;

put $$deferred;

unset $$deferred;
```

11.6 The Putvars Statment

putvars has already been used in previous examples to examine the variables in an event. Putvars is a special statement that prints all the values for a given category of variables. The first argument to putvars is the variable category to print. The categories are shown in table 11.1 on page 99.

After the first argument, putvars behaves just like a put, except that it will execute once for each variable in the chosen category. It is an implicit loop where the name of the variable is placed in a special variable _name_, and the value of that variable is placed in _value_. A very common use of putvars looks like this.

```
putvars style _name_ ' : ' _value_ nl;
```

11.7 Bringing it all together

It's quite easy to create a tagset to test all of these things. The initialize event is perfect for this sort of thing. The doc event should be used with SAS 9.0 or earlier. The code example on page 100 combines the examples in this chapter into one event. The output is shown in 11.2 on page 101.

Table 11.1: Putvars: Variable Categories

Event	Data and Metadata variables
Style	Style variables
Memory	Memory variables created with set or eval statements
Stream	Stream variables created with the open statement
Dynamic	Dynamic variables created within SAS.
Lists	Any list variable.
Dictionaries	Any Dictionary variable.

Listing 11.1: Using Variables

```
proc template;

    define tagset tagsets.variables;

        /* define event doc; */
        define event initialize;

            set $test  operator ' ran this job.';
            set $test2 $test ' On this date: ' date;
            set $test3 compress($test2, ':.');

            put 'Simple string Variables' nl;
            putvars memory _name_ ': ' _value_ nl;

            set $test[]  operator ' ran this job.';
            set $test[] $test[1] ' On this date: ' date;
            set $test[] compress($test[2], ':.');

            put nl 'A List' nl;
            putvars $test _name_ ': ' _value_ nl;

            set $test[] 'Entry 1';
            set $test[] 'Entry 2';
            set $test[] 'Entry 3';

            set $test[2] 'The New 2';

            put nl 'The same List' nl;
            put 'Test now has a length of ' $test nl;

            putvars $test _name_ ': ' _value_ nl;

            put nl 'Print the last 3 in reverse' nl;
            /* print them in reverse order */
            put $test[-1] nl;
            put $test[-2] nl;
            put $test[-3] nl;

            /* delete the first entry */
            put 'Deleting test[1]' nl;
            unset $test[1];

            put 'Test now has a length of ' $test nl;

            putvars $test _name_ ': ' _value_ nl;

            /* delete the whole thing */

            unset $test;
```

```
            put 'Test has a length of ' $test nl;

            set $test['one'] operator ' ran this job.';
            set $test['one plus'] $test['one'] ' On this date: ' date;
            set $test['one sentence'] compress($test['one plus'], ':.') '.';

            put nl 'A Dictionary' nl;
            put 'Test has a length of ' $test nl;

            putvars $test _name_ ': ' _value_ nl;

            put nl 'Print a dictionary entry with a key' nl;
            set $key 'one sentence';
            put $test[$key] nl;

            unset $test;

            eval $count 1;
            eval $count $count+1;
            eval $position index($key, 's');
            eval $string1 substr($key, $position);
            set $string2 substr($key, $position);

            put nl 'Miscellaneous variables' nl;
            putvars memory _name_ ': ' _value_ nl;

            open deferred;
                put 'we will print this later. Goodbye.' nl;
            close;
            put nl 'This will print now.' nl;

            put $$deferred;

            unset $$deferred;

        end;
    end;
run;

ods tagsets.variables file='test.txt';
ods tagsets.variables close;
```

Listing 11.2: Output: Using Variables

```
Simple string Variables
test: eric ran this job.
test2: eric ran this job. On this date: 2003-10-26
test3: eric ran this job On this date 2003-10-26

A List
```

```
: eric ran this job.
: eric ran this job. On this date: 2003-10-26
: eric ran this job On this date 2003-10-26

The same List
Test now has a length of 6
: eric ran this job.
: The New 2
: eric ran this job On this date 2003-10-26
: Entry 1
: Entry 2
: Entry 3

Print the last 3 in reverse
Entry 3
Entry 2
Entry 1
Deleting test[1]
Test now has a length of 5
: The New 2
: eric ran this job On this date 2003-10-26
: Entry 1
: Entry 2
: Entry 3

A Dictionary
Test has a length of 3
one: eric ran this job.
one plus: eric ran this job. On this date: 2003-10-26
one sentence: eric ran this job On this date 2003-10-26.

Print a dictionary entry with a key
eric ran this job On this date 2003-10-26.

Miscellaneous variables
test2: eric ran this job. On this date: 2003-10-26
test3: eric ran this job On this date 2003-10-26
key: one sentence
count: 2
position: 5
string1: sentence
string2: sentence

This will print now.
we will print this later.  Goodbye.
```

11.8 Summary

This chapter uses a fairly simplistic tagset that does nothing but show the various forms of variables that can be created. Strings are by far the most common type of variable. Occasionally only a numeric variable will do what is needed. It may not be apparent, but Dictionaries and lists can be extremely useful for simplifying problems and creating new possibilities.

Chapter 12

Procedural controls

The tagset's if syntax has already been used in previous examples. With SAS 8.2 and 9.0 if statements were kept to a few simple choices. This simple if statement is still the most efficient and most frequently used type of conditional programming that tagsets have. With SAS 9.1 if statments have been extended with where clauses and new statements to provide fully featured procedural controls. The examples in this chapter will explain the if syntax and the procedural control statements that complement them.

12.1 Simple If's

An if can be placed at the end of almost any tagset statement[1] . But it must be preceded by a /. Usually an *if* follows the /. But that is optional. In fact, the keywords *when* and *where* are also allowed. But it's all personal preference since none of the words mean anything. There are two cases where the keyword following the / is important. *Breakif* and *While* cause the if to do more than a simple test.

12.1.1 Built in tests

These four tests are very tightly integrated and are therefore faster than any other type of test. Any of these can be preceded with a ! or ŵhich indicates negation of the result. The ! can only be used for 'not' at the very beginning of an if, while ĉan be used at any time. Table 12.1 on page 106 shows the forms of the builtin if tests. One very important distinction between these if tests is that the variables in the tests do not have to be defined. If a variable doesn't exist the tests will evaluate to false. This is very convenient for boolean variables. Set the variable to some string for true, and unset it for false. On the other hand, referencing a non-existant variable in a where clause frequently causes runtime errors.

If a variable in a where clause is not defined then an error will result.

The following example uses each of these tests in use.

```
define event initialize;
   set $is_eric 'True' /if cmp(operator, 'eric');
   put 'Hello Eric' nl /if $is_eric;
   put 'Hello ' operator nl /if ^$is_eric;
   put 'RIC' nl /if contains(operator, 'ric');
```

[1] eval and done are the only statements that cannot have an if

Table 12.1: Built in 'if' tests

Cmp	Case insensitive compare of two string values.
Exists	Check if all values have a length or are non-zero
Any	Check if any values have a length or are non-zero
Contains	Case sensitive search for string inside a string.
Variable	Is the string length or numeric value.

```
    put 'This might be Eric' nl /if any(operator, $is_eric);
    put 'This is Eric' nl /if exists(operator, $is_eric);
end;
```

12.2 Where Clauses

In addition to the simple if conditions above if tests can also use where clauses. It turns out that they are usually used when doing numeric comparisons, but some problems reguire more complex if's. The where processing provides that.

```
define event initialize;
    eval $count 1;
    put 'Count is > 0' nl /if $count > 0;

    eval $max 10;
    put 'max > Count > 0 ' nl /if $count > 0 and $count < $max;
end;
```

12.3 Break

Break is a statement just like put or set. When encountered the program flow exits the event immediately. This is useful for skipping the end of an event when the remaining statements are not desired. Here is an example based on the initialize event from the example in in figure 7.8 on page 49.

```
define event initialize;
    set $delimiter tagset_alias;
    break /if $delimiter;

    set $delimiter csv_delimiter /if !$delimiter;
    set $delimiter ',' /if !$delimiter;
end;
```

This is more efficient when tagset_alias is used because only one if is executed instead of two. But it is less efficient if neither tagset alias or the csv_delimiter macro variable is set because all three if's will have to execute. Efficiency is not so important in this case, but it can be very important if the event is used very frequently. A poorly coded data event can have a significant impact on performance.

12.4 Breakif

Breakif is one of the two special keywords that can be used to define an if test on a tagset statement. When looking through the tagset code, there are many times when it is desirable to do something and then break if a certain condition is met. With breakif it is possible to write this with one line instead of two. That means one test evaluation instead of two. Breakif[2] can be quite useful in these situations because it combines break with the execution of another command based on an if. Using the same example using breakif looks like this:

```
define event initialize;
   set $delimiter tagset_alias  /breakif tagset_alias;
   set $delimiter csv_delimiter /breakif csv_delimiter;
   set $delimiter ',';
end;
```

This is a nice, efficient, and readable way to control the program flow. As soon as any one of the if's is true the delimiter will be set and the event will be exited.

12.5 Do blocks

Do blocks[3] provide a way to group several event statements together under one condition. do blocks start with a Do statement, end with a Done statement and may have any number of else statements in between. Using the same example with do else looks like this.

```
define event initialize;

   do /if tagset_alias;
        set $delimiter tagset_alias;

   else /if csv_delimiter;
        set $delimiter csv_delimiter;

   else;
        set $delimiter ',';
   done;

end;
```

This is the most efficient construct so far. but can be refined further by eliminating one more if. If tagset_alias has no value, the set does nothing, so putting an if around it isn't really an improvement. It comes down to readability and personal taste.

```
define event initialize;

   set $delimiter tagset_alias;

   do /if !$delimiter;
        set $delimiter csv_delimiter;
```

[2]Available in SAS 9.1 and later
[3]Available in SAS 9.1 and later

```
        else;
            set $delimiter ',';
        done;

    end;
```

12.6 Do While Loops

Using the do statement it is also possible to create loops. The behavior of the do block is changed to do looping with the help of another special keyword. Like *breakif*, *while*[4], has special meaning. it causes the do block to loop as long as the while test is true. A Do /while block can also have an else, or even multiple else /if's. If the initial while test is false, execution will fall to the else. But if the while block executes at all then the else will not be executed.

```
proc template;
define tagset tagsets.dowhile;
    default_event = 'count';

    define event initialize;
        eval $count 1;

        do /while $count <3;
            put "Count < 3 " $count nl;
            eval $count $count + 1;
        else;
            put "Else from: Count < 3: " $count nl;
        done;

        do /while $count <3;
            put "Second Count < 3:" $count nl;
            eval $count $count + 1;
        else;
            put "Second Else: Count >= 3: " $count nl;
        done;

    end;
end;
run;

ods tagsets.dowhile file="dowhile.txt";
ods tagsets.dowhile close;
```

The output shows a count of 1 and 2. The second loop fails to execute and displays the message from the else. The output follows.

```
Count < 3 1
Count < 3 2
```

[4]The while keyword is only valid on the Do statement

```
Second Else: Count >= 3: 3
```

To round out the looping controls there are stop and continue statements. Stop will cause the loop to stop, program flow will pick up at the done. Continue will cause program flow to start back at the top of the loop again. Break behaves as always and causes the entire event to be aborted. It is very easy to make infinite loops, be careful.

```
eval $i 0;

put "i is " $i nl;

put "Going into a Loop to 10" nl;
put "Continue at 5" nl;
put "stop at 8" nl;

do /while $i < 10;

    eval $i $i+1;

    continue /if $i eq 5;
    stop     /if $i eq 8;

    put "I is " $i nl;

else;
    put "do this if i started out > 10" nl;

done;
```

12.7 Iterating through Dictionaries

Using do loops, the capability of iterating through a dictionary of values would be a very nice thing to do. Looping through a list is easy, all that is needed is a counter. But dictionary's are not so easy. To loop through a dictionary there are iterate and next statements. Iterate and next work similarly to putvars in that they populate the special variables _name_ and _value_. Iterate sets up the iterator for us and populates the first name and value into _name_ and _value_ respectively. From then on, only the next statement is used. Of course an iterator will work on a list too, but there won't be a _name_. These examples are awfully simple, putvars could have done this in one line. The output can be seen in figure 12.2 on page 112.

```
set $dogs['Ernie'] 'Was a great dog';
set $dogs['Pola'] 'Was really sweet.';
set $dogs['Arkas'] 'Is a big dog';
set $dogs['Luna'] 'Is super sweet';

iterate $dogs;
do /while _value_;
    put "Key: " _name_ " Value: " _value_ nl;
    next $dogs;
```

```
done;

set $dog_keys[] 'Ernie';
set $dog_keys[] 'Pola';
set $dog_keys[] 'Arkas';
set $dog_keys[] 'Luna';

iterate $dog_keys;
do /while _value_;
    put "Key: " _name_ " Value: " _value_ nl;
    next $dogs_keys;
done;
```

12.8 Bringing it all together

The code example on page 111 shows everything covered in this chapter in one event so that it can be easy to see how things work.

Listing 12.1: Using Procedural controls

```
proc template;

    define tagset tagsets.controls;

        define event bif;
            put 'Hello Eric' nl /break_if $is_eric;
            put 'Greetings ' operator nl;
        end;

        define event initialize;
            set $is_elmo 'True' /if cmp(operator, 'elmo');
            set $is_eric 'True' /if cmp(operator, 'eric');
            put 'Hello Eric' nl /if $is_eric;
            put 'Greetings ' operator nl /if ^$is_eric;
            put 'RIC' nl /if contains(operator, 'ric');
            put 'This might be Eric' nl /if any(operator, $is_eric);
            put 'This is Eric' nl /if exists(operator, $is_eric);

            trigger bif;

            do /if $is_eric;
                put 'Go away' nl;

            else /if $is_elmo;
                put "What's up?" nl;

            else;
                put 'Hello ' operator ' How are you?' nl;
            done;


            eval $i 0;

            put nl 'i is ' $i nl;

            put "Going into a Loop to 10" nl;
            put "Continue at 5" nl;
            put "stop at 8" nl;

            do /while $i < 10;

                eval $i $i+1;

                continue /if $i eq 5;
                stop     /if $i eq 8;

                put 'I is ' $i nl;
```

```
            else;
                put 'do this if i started out > 10' nl;

            done;

            set $dogs['Ernie'] 'Was a great dog';
            set $dogs['Pola'] 'Was really sweet.';
            set $dogs['Arkas'] 'Is a big dog';
            set $dogs['Luna'] 'Is super sweet';

            put nl 'iterating on a dictionary' nl;
            iterate $dogs;
            do /while _value_;
                put "Key: " _name_ " Value: " _value_ nl;
                next $dogs;
            done;

            set $dog_keys[] 'Ernie';
            set $dog_keys[] 'Pola';
            set $dog_keys[] 'Arkas';
            set $dog_keys[] 'Luna';

            put nl 'iterating on a list' nl;
            iterate $dog_keys;
            do /while _value_;
                put "Key: " _name_ " Value: " _value_ nl;
                next $dog_keys;
            done;

        end;
    end;
run;

ods tagsets.controls operator='eric' file='test.txt';
ods tagsets.controls close;
```

Listing 12.2: Output: Control statements Example

```
Hello Eric
RIC
This might be Eric
This is Eric
Hello Eric
Go away

i is 0
Going into a Loop to 10
Continue at 5
stop at 8
I is 1
I is 2
```

```
I is 3
I is 4
I is 6
I is 7

iterating on a dictionary
Key: Ernie Value: Was a great dog
Key: Pola Value: Was really sweet.
Key: Arkas Value: Is a big dog
Key: Luna Value: Is super sweet

iterating on a list
 Value: Ernie
 Value: Pola
 Value: Arkas
 Value: Luna
```

12.9 Summary

The procedural capabilities of tagsets have grown exponentially with the release of SAS 9.1. Simple if's are still the most common and efficient type of if. But with the addition of where clauses the capabilities are almost endless. Single line, immediate, if's are still quite common, breakif enhances those abilities for readability and efficiency. Do blocks and while provide even more readability, efficiency and control.

Chapter 13

Trigger Happy

ODS triggers, or requests, many events internally. It is also possible to trigger events from other events. This chapter is going to explain the trigger statement and how to use it.

13.1 Simple Triggers

In previous examples, the trigger statement has been used for simple cases where the triggered event is a simple event without a start or finish. This type of use is very straight forward and intuitive.

There is a special variable that will always have the name of the current triggered event. Trigger_name will be empty if the current event was triggered from within ODS. The event_name variable will always have the name of the initial ODS event. This variable is useful in explaining how all of this works.

The following example is the simple form of trigger that has been used previously. For convenience the doc event is being used for this example. The do_stuff event is being triggered from both the start and finish of the doc event. The message from the do_stuff event will print twice, as expected.

```
define event doc;
   start:
       trigger do_stuff;
   finish:
       trigger do_stuff;
end;

define event do_stuff;
     put 'This is the ' trigger_name ' Event' nl;
end;
```

13.2 Events with a state

Events can have what is called a state. In the example above, the doc event has two states. The first time it is triggered it's state is start. The second time it is in the finish state. The do_stuff event is stateless. It does the same thing always. The 'state' variable indicates the current state. It's value will be either start or finish.

This is important because triggered events preserve the state of the calling event. When a trigger statement is in the start section of an event, the start section of the triggered event is executed. Likewise, when the trigger statement is in the finish section, the finish section of the triggered event is executed. The following example defines a start and finish in the triggered event. The result is that each message is printed once.

```
define event doc;
   start:
       trigger do_stuff;
   finish:
       trigger do_stuff;
end;

define event do_stuff;
    start:
        put state ': ' trigger_name ' Event' nl;
    finish:
        put state ': ' trigger_name ' Event' nl;
end;
```

But what if the triggered event only defined a start or finish? Then it would only execute for the state section defined. The following example will only print one message, once.

```
define event doc;
   start:
       trigger do_stuff;
   finish:
       trigger do_stuff;
end;

define event do_stuff;
    finish:
        put state ': ' trigger_name ' Event' nl;
end;
```

There are times when this behavior is not desirable. Events are commmonly defined with a start and finish that would be placed around another event or a put. There is a third argument to the trigger statement that allows this. After the event name, start or finish can be specified. It may be a good idea to always specify start and finish just to clarify what the code is doing. In this example indention is added to help clarify the nesting of starts and finishes. The example in figure 13.1 on page 116 shows all of the various trigger behaviors. The output is shown if figure 13.2 on page 117.

Listing 13.1: Triggers Example

```
proc template;

   define tagset tagsets.triggers;
       indent = 4;

       define event doc;
          start:
              trigger do_stuff;
```

```
            trigger do_stuff start;
            trigger inbetween;
            trigger do_stuff finish;
        finish:
            trigger do_stuff start;
            trigger inbetween;
            trigger do_stuff finish;
            trigger do_stuff;
    end;

    define event do_stuff;
        start:
            put state ': ' trigger_name nl;
            ndent;
            trigger more_stuff;
        finish:
            trigger more_stuff;
            xdent;
            put state ': ' trigger_name nl;
    end;

    define event more_stuff;
        start:
            put state ': ' trigger_name nl;
    end;

    define event inbetween;
        put state ': ' trigger_name nl;
    end;

end;
run;

ods tagsets.triggers file='test.txt';
ods tagsets.triggers close;
```

Listing 13.2: Output: Triggers Example

```
start: do_stuff
    start: more_stuff
    start: do_stuff
        start: more_stuff
        start: inbetween
    finish: do_stuff
    start: do_stuff
        start: more_stuff
        finish: inbetween
    finish: do_stuff
finish: do_stuff
```

13.3 summary

Triggered events are great way to compartmentalize commonly used code. When combined with if and breakif they are also a good way to control the program flow. If your events are getting complicated with lots of do / elses then it's probably time to create some new events and trigger them. Triggered events are just as fast as inline code, so performance is not an issue. Readability and maintenance of the tagset is.

Previous to SAS 9.1 triggers were the only way to control the program flow. Many of the tagsets still have code that reflect this programming style. Triggered events are overly complex in some cases.

Chapter 14

Data Step Functions

Data step functions provide important capabilities to tagsets. Many solutions are dependent on string manipulation, formatting and number conversion. Without the functions, these solutions would be very difficult or impossible. This chapter will explain the usage of data step functions in tagsets.

14.1 Set and Put statements

The simplest and most efficient use of data step functions is in the put and set statements. The only limitation to using functions in these statements is that they cannot be nested. This is generally not a disadvantage.

Searching through tagsets released by SAS, the most popular functions are tranwrd, index, substr, scan, strip and the perl regular expressions. These functions return values will be strings or will be converted to strings when used in the set and put statements.

14.2 The Eval Statement

If a numeric return value is desired then the function must be used in an eval statement. The eval statement also allows for nested functions. This is because eval supports all valid where clauses which allow this.

The following example is taken directly from the html tagsets released with SAS 9.1. This code shows eval being used to capture the location of the spaces in the url. Everything else uses a set statement to store string values. It would be possible to combine multiple statements into one, if the set statements were replaced with an eval statement that nested the two datastep functions.

This particular event will only happen once per file so that may be a reasonable thing to do. But it should be noted that over use of eval and where clauses can cause tagsets to be slow.

```
define event urlLoop;
    eval $space_pos index($urlList, " ");

    do /while $space_pos ne 0;

        do /if $space_pos ne 0;
            set $current_url substr($urlList,1,$space_pos);
            set $current_url trim($current_url);
```

```
            done;

            trigger link;

            set $urlList substr($urlList,$space_pos);
            set $urlList strip($urlList);

            eval $space_pos index($urlList, " ");
        done;
        /* when space_pos is 0 it's either the only link or the last link */
        set $current_url $urlList;
        trigger link;
    end;
```

The purpose of this event is to take a stylesheet url that has multiple urls separated by spaces. When that occurs, this event will pull apart the urls and create multiple link tags in the html. An ods statement that uses this feature would look like this.

```
ods html file='test.html' stylesheet='test.css'(url='test.css test2.css');
```

This event is a nice example of several functions, but there is a much simpler way to do this. The scan function does all the work of index, substr, strip, and trim.

```
define event urlLoop;
    eval $count 1;
    set $current_url scan($urlList, 1, " ");

    /*-------------------------------------------------------------eric-*/
    /*-- scan returns a space when it fails.                        --*/
    /*-------------------------------------------------------------20Nov03-*/
    do /while !cmp($current_url, ' ');

        trigger link;

        eval $count $count + 1
        set $current_url scan($urlList, $count, " ");
    done;
end;
```

14.2.1 number conversions

Converting strings to numbers is one of the most common things tagsets need to do. It seems to happen a lot. Using numbers to indicate a column or row in a table is a common desire. Often these numbers come in the form of a macro variable. The inputn function is the way to convert these strings to numbers. Inputc also works. But the input function does not. Using input will result in a runtime error. Following code shows common misuse and usage.

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

```
            mvar some_number;

            define event initialize;
               eval $inputn_value  inputn(some_number, '3.')+5;
               set $set_inputn  inputn(some_number, '3.');
               eval $input_value  input(some_number, 3.)+1;
               put ":" putc("hello", '$', 15) ":" nl;
               put ":" put("hello", '$15') ":" nl;
               putvars mem _name_ ":" _value_ nl;
            end;
         end;
      run;

      %let some_number=10;

      ods tagsets.input file="input.txt";
      ods tagsets.inpuc close;
```

The log looks like this:

```
NOTE: Overwriting existing template/link: Tagsets.Input
NOTE: TAGSET 'Tagsets.Input' has been saved to: SASUSER.TEMPLAT
16          run;
NOTE: PROCEDURE TEMPLATE used (Total process time):
      real time           0.12 seconds
      cpu time            0.12 seconds



17
18         %let some_number=10;
19
20         ods tagsets.input file="input.txt";
WARNING: Function not found: input
WARNING: Function not found: put
NOTE: Writing TAGSETS.INPUT Body file: input.txt
```

And the Output looks like this.

```
:hello          :
:
inputn_value:15
set_inputn:10
input_value:11
```

Obviously, some datastep functions do not work. Input gives a warning but seems to work anyway. Put doesn't work at all. But putc does. It's probably a good idea to stay away from functions that give warnings.

14.3 advanced usage and debugging

String manipulation or using inputn to convert a string to a number are easy. But sometimes it is not obvious when to use eval or set. The choice comes down to the return value of the function and if it is needed as a parameter to another function later on.

14.3.1 File I/O

This next example uses datastep functions to read in a file. It is important that some of the return codes be integers so that they can be used as parameters later on. The key to understanding this code is using putlog to print the return values from the functions. The return values can be very revealing, If the file is not found the return from the filename function will be missing, or '.'. if close is used instead of fclose the return value wil be a large number. If everything comes back with 0's all is well. Except the file identifier, it should be a unique number. It must be an integer to be used by fread and fget functions. That means that eval must be used when the file identifier is created.

Another trick that this example uses is the creation of a variable with a fixed length. Data step functions always want fixed length variables. But tagsets do not have any. The solution is to create a variable full of spaces. The functions will not change the length of it, they will only put values in it.

```
/*-------------------------------------------------------------------------*/
/*-- Reading a file in using datastep functions.  This example            --*/
/*-- comes straight out of the online documentation                        --*/
/*-- for fread().                                                          --*/
/*-------------------------------------------------------------------------*/
proc template;
    define tagset tagsets.readfile;
        parent=tagsets.html4;
        embedded_stylesheet=no;

        mvar infile;


        /*-------------------------------------------------------------eric-*/
        /*-- The files should be given as a list with spaces between them.--*/
        /*-----------------------------------------------------------8Nov 03-*/
        define event initialize;
            set $filename 'default.txt';

            set $filrf "myfile";
            trigger readfile;
        end;

        define event readfile;

            /*-----------------------------------------------------------*/
            /*-- Set up the file and open it.                          --*/
            /*-----------------------------------------------------------*/
            putlog "Reading in file: " $filename;

            eval $fid 0;

            eval $rc filename($filrf, $filename);
            putlog "Filename Return Code" ":" $rc;

            eval $fid fopen($filrf);
```

```
            do /if missing($fid);
                putlog "Error: File Not Found, " $filename;
                break;
            done;

            putlog "File ID" ":" $fid;

            /*-------------------------------------------------------------*/
            /*-- datastep functions  will bind directly to the            --*/
            /*-- variable space as it exists.                             --*/
            /*--                                                          --*/
            /*-- Tagset variables are not like datastep                   --*/
            /*-- variables but we can create a big one full               --*/
            /*-- of spaces and let the functions write to it.             --*/
            /*--                                                          --*/
            /*-- This creates a variable that is 200 spaces so           --*/
            /*-- that the function can write directly to the              --*/
            /*-- memory location held by the variable.                    --*/
            /*-- in VI, 200i<space>                                       --*/
            /*-------------------------------------------------------------*/
            set $file_record  "
";

            /*-------------------------------------------------------------*/
            /*-- Loop over the records in the file                        --*/
            /*-------------------------------------------------------------*/
            do /if $fid > 0 ;

                do /while fread($fid) = 0;

                    set $rc fget($fid,$file_record ,200);
                    putlog "FGet rc" ":" $rc;

                    putlog "Record:" trimn($file_record);
                done;
            done;

           /*-------------------------------------------------------------*/
           /*-- close up the file.  set works fine for this.             --*/
           /*-------------------------------------------------------------*/

            set $rc fclose($fid);
            putlog "close rc" ":" $rc;
            set $rc filename($filrf);
            putlog "filename rc" ":" $rc;

        end;
    end;
run;
```

```
ods tagsets.readfile file="file.txt";
ods tagsets.readfile close;
```

With simple four line input file, the output to the log generated by this tagset looks like this.

```
NOTE: Writing TAGSETS.READFILE Body file: file.txt
Reading in file: default.txt
Filename Return Code:0
File ID:1
FGet rc:0
this is a test
FGet rc:0
the quick brown
FGet rc:0
fox jumped over
FGet rc:0
the lazy dog.
close rc:0
filename rc:0
83          ods tagsets.readfile close;
```

14.3.2 Perl Regular Expressions

Another complex example comes from the perl regular expressions. The compiled regular expression is a unique integer that is used when getting the substrings from the matched string. The following example pulls apart the value of the papersize option for use in an xml document. The papersize option is extremely variable in it's format. That makes the perl regular expressions a good choice for solving this problem.

Listing 14.1: Using Perl Regular Expressions

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

     define event initialize;
         set $papersize getoption('PAPERSIZE');

         put "Papersize" ":" $papersize nl;

         /* tranwrd just makes the regex easier. Get rid of optional quotes */
         set $papersize tranwrd($papersize, '"', " ");
         set $papersize tranwrd($papersize, "'", " ");

         /* could be centimeters, could be quoted, or not...
            default is supposedly inches but could be installation
            dependent.
            (8in 11in);
            ('8in', '11in');
            ("8in", "11in");
            ("8", "11");
         */
```

```
        /*-------------------------------------------------------------------eric-*/
        /*-- The compiled regular expression is a unique integer.
--*/
        /*-------------------------------------------------------------21Nov03-*/
         eval $re prxparse('( *([0-9]+) *(IN|CM)* *[,]+ *([0-9]+) *(IN|CM)*.*)') ;
         put "RE is " ":" $re nl;

         eval $match prxmatch($re, $papersize);
         put "MATCH is " ":" $match nl;

        /*-------------------------------------------------------------------eric-*/
        /*-- Get the width and the width unit
--*/
        /*-------------------------------------------------------------21Nov03-*/
         set $pwidth prxposn($re, 1, $papersize) ;
         set $pwidth_unit prxposn($re, 2, $papersize) ;

         set $pwidth_unit lowcase($pwidth_unit) /if $pwidth_unit;
         set $pwidth_unit "in" /if !$pwidth_unit;

        /*-------------------------------------------------------------------eric-*/
        /*-- Get the height and the height unit.
--*/
        /*-------------------------------------------------------------21Nov03-*/
         set $pheight prxposn($re, 3, $papersize) ;
         set $pheight_unit prxposn($re, 4, $papersize) ;

         set $pheight_unit lowcase($pheight_unit) /if $pheight_unit;
         set $pheight_unit "in" /if !$pheight_unit;

         /* Only if they are non-zero */
         put ' height="' $pheight $pheight_unit '"' nl /if $pheight;
         put ' width="' $pwidth $pwidth_unit '"' nl /if $pwidth;

         unset $papersize;
         unset $re;
         unset $pwidth;
         unset $pwidth_unit;
         unset $pheight;
         unset $pheight_unit;
      end;

  end;
run;

options papersize=("5in" , 10cm);

ods tagsets.regex file="regex.txt";
ods tagsets.regex close;
```

The output from this tagset looks like this.

```
Papersize:("5IN" , 10CM)
RE is :1
MATCH is :2
 height="10cm"
 width="5in"
```

14.4 Summary

Date step functions are another versatile tool available to the tagset programmer. Functions are simple to use and provide great versatility and power. Many things would not be possible without them. Functions do need to be used with care, checking the return values from them is always a good idea.

Part IV

Intermediate Examples

Chapter 15

Styles and Tagsets, A perfect match

Understanding how to use ODS styles with tagsets can simplify many problems. Styles can help make a tagset more versatile and reusable. They can also help to make the output less verbose and more compartmentalized. This chapter will demonstrate the basics of using styles with tagsets. The best way to start is with a simple tagset designed specifically to show how styles work.

15.1 Starting Simple

The easiest way to examine any problem is to break it down as much as possible. It is easy to see how styles work with a tagset that has one event. Any event will do. The table event is a good place to start. It is a familiar event, and tables have some interesting style attributes.

A single putvars statement can do all the work of revealing how styles work.

Listing 15.1: Show table style

```
proc template;

    define tagset tagsets.showstyle;

        define event table;
            putvars style _name_ ' : ' _value_ nl;
        end;

     end;
run;

ods tagsets.showstyle file='style.txt';
proc print data=sashelp.class; run;
ods tagsets.showstyle close;
```

The output in figure 15.2 shows all the style attributes for the print procedure's table event.

Listing 15.2: Output: The default table style

```
FRAME : BOX
RULES : GROUPS
HTMLCLASS : Table
CELLSPACING : 1
CELLPADDING : 7
FRAMEBORDER : auto
CONTENTSCROLLBAR : auto
BODYSCROLLBAR : auto
```

But where did these come from? ODS has a sytle called default that it uses if no other style is specified on the ods statement. The default style has a style element called table which has these attributes defined. There is an automatic association between the table event and the table style. In fact, most events have a style that is automatically associated with them. By default, the value of HTMLCLASS will be the name of the style element defined in the style template.

It is noticable is that there are no colors or fonts, yet those things must be specified, how could they not be? In fact, they are defined in the style template. But they don't show up here because, by default, tagsets screen out most of the values except during a few special events that are used to create style definitons for the output. The most common manifestation is an HTML Cascading stylesheet definition. If all those style attributes are defined elsewhere then there is no reason to repeat them on every event. Setting the pure_style attribute to yes will change this behavior so that all the style attributes are available all of the time.

It could be argued that cellpadding and cellspacing belong in the stylesheet, not here in the table event. That is technically true. But some less capable browsers cannot render tables correctly using those stylesheet definitions. So cellpadding and cellspacing remain here with frame and rules until the web browsers can get it right. This is mostly an HTML issue since these things are not a problem in other markup languages like latex, troff and XML.

The next step is to create a new style, and add a new event to expose this behavior. Figure 15.3 on page 130 shows the new code. There is a new style called showstyle which contains one style element. The table style element defines a font and colors, as well as some table specific attributes. The tagset has a new event, style_class. Style_class is the event used to create each style definition in the output. Most events are directed at the body file by default. But style_class is one of those events that is not. By default style_class is directed at the stylesheet file. In order to see the output from the style_class event a stylesheet file was specified on the ods statement. Instead of adding a stylesheet file, we could have also just set the embedded_stylesheet attribute to yes. The new style must also be specified. The output from the body file in figure 15.4 on page 131 hasn't changed much, it still shows the same variables from before. But the output in the stylesheet file in figure 15.5 shows much more.

Listing 15.3: Show table style

```
proc template;

    define style styles.showstyle;

        style table /
          Font = ("Verdana, Helvetica, sans-serif", 4, bold)
          Foreground = black
          background = White
          bordercolor = green
          cellspacing = 1
```

```
            cellpadding = 3
            rules = groups
            frame = box
          ;
      end;

      define tagset tagsets.showstyle;
          indent=4;

          define event style_class;
             trigger put_style;
          end;

          define event table;
             trigger put_style;
          end;

          define event put_style;
              put nl 'Event: ' Event_Name nl;
              ndent;
              putvars style _name_ ' : ' _value_ nl;
              xdent;
          end;

       end;
run;

ods tagsets.showstyle style=showstyle
                      file='style.txt'
                 stylesheet='style2.txt';

proc print data=sashelp.class; run;
ods tagsets.showstyle close;
```

Listing 15.4: Output: The table event style: style.txt

```
Event: table
    FRAME : box
    RULES : groups
    HTMLCLASS : table
    CELLSPACING : 1
    CELLPADDING : 3
    FRAMEBORDER : auto
    CONTENTSCROLLBAR : auto
    BODYSCROLLBAR : auto
```

The stylesheet file contains much more information about the style.

Listing 15.5: Output: The table class style: style2.txt

```
Event: style_class
```

```
    FRAME : box
    RULES : groups
    FONT_FACE : Verdana, Helvetica, sans-serif
    HTMLCLASS : table
    CELLSPACING : 1
    CELLPADDING : 3
    FONT_SIZE : medium
    BACKGROUND : #FFFFFF
    FOREGROUND : #000000
    BORDERCOLOR : #008000
    FONT_WEIGHT : bold
    FONT_STYLE : normal
    FONT_WIDTH : normal
    FRAMEBORDER : auto
    CONTENTSCROLLBAR : auto
    BODYSCROLLBAR : auto
```

15.1.1 Embedded styles

Having all of the style definitions in a separate file can be nice. But there are times when a separate stylesheet file is inconvenient. This example is one of those times. Tagsets allow for this with an attribute called 'embedded_stylesheet'. By default the value of this attribute is false. Turning it on simplifies the previous example because a separate stylesheet file is no longer needed. One can still be specified, and everything will work just like the last example. But if it isn't, hen the style_class events are directed at the body file. In this example inheritance is also used. This makes it easy to see what has changed from one tagset to the next. For the purposes of this book it also makes the examples much shorter. The new tagset is in figure 15.6. The corresponding output is in figure 15.7.

Listing 15.6: Show table style

```
proc template;
    define tagset tagsets.showstyle1;
        parent=tagsets.showstyle;
        embedded_stylesheet = yes;
     end;
run;

ods tagsets.showstyle1 style=showstyle file='style.txt';
proc print data=sashelp.class; run;
ods tagsets.showstyle close;
```

Now all the output shows up in the body file.

Listing 15.7: Output: The table event style

```
Event: style_class
    FRAME : box
    RULES : groups
    FONT_FACE : Verdana, Helvetica, sans-serif
```

```
    HTMLCLASS : table
    CELLSPACING : 1
    CELLPADDING : 3
    FONT_SIZE : medium
    BACKGROUND : #FFFFFF
    FOREGROUND : #000000
    BORDERCOLOR : #008000
    FONT_WEIGHT : bold
    FONT_STYLE : normal
    FONT_WIDTH : normal
    FRAMEBORDER : auto
    CONTENTSCROLLBAR : auto
    BODYSCROLLBAR : auto

Event: table
    FRAME : box
    RULES : groups
    HTMLCLASS : table
    CELLSPACING : 1
    CELLPADDING : 3
    FRAMEBORDER : auto
    CONTENTSCROLLBAR : auto
    BODYSCROLLBAR : auto
```

15.2 Styles of your own choosing

These examples have been using the style names that ODS decided to use. But it is possible to create and use style elements using any name. This can be done by using the the event's style attribute. The code in figure 15.8 on page 133. Adds a new style element, darkbox, and then uses it in the table event. The output in figure 15.9 on page 134 shows two style class events and one table event which reveals the new style. Note that any new style elements in the ODS style result in another request of the style_class event.

Listing 15.8: Show table style

```
proc template;

    define style styles.showstyle;

        style table /
          Font = ("Verdana, Helvetica, sans-serif", 4, bold)
          Foreground = black
          background = White
          bordercolor = green
          cellspacing = 1
          cellpadding = 3
          rules = groups
          frame = box
        ;
```

```
        style darkbox /
          Font = ("Times", 6, bold)
          Foreground = black
          background = White
          bordercolor = Black
          cellspacing = 0
          cellpadding = 1
          rules = none
          frame = box
        ;
    end;

    define tagset tagsets.showstyle2;
        parent=tagsets.showstyle1;

        define event table;
           style=darkbox;
           trigger put_style;
        end;
    end;
run;

ods tagsets.showstyle2 style=showstyle file='style.txt';
proc print data=sashelp.class; run;
ods tagsets.showstyle close;
```

Listing 15.9: Output: Specified Style

```
Event: style_class
    FRAME : box
    RULES : none
    FONT_FACE : Times
    HTMLCLASS : darkbox
    CELLSPACING : 0
    CELLPADDING : 1
    FONT_SIZE : x-large
    BACKGROUND : #FFFFFF
    FOREGROUND : #000000
    BORDERCOLOR : #000000
    FONT_WEIGHT : bold
    FONT_STYLE : normal
    FONT_WIDTH : normal
    FRAMEBORDER : auto
    CONTENTSCROLLBAR : auto
    BODYSCROLLBAR : auto

Event: style_class
    FRAME : box
    RULES : groups
    FONT_FACE : Verdana, Helvetica, sans-serif
```

```
    HTMLCLASS : table
    CELLSPACING : 1
    CELLPADDING : 3
    FONT_SIZE : medium
    BACKGROUND : #FFFFFF
    FOREGROUND : #000000
    BORDERCOLOR : #008000
    FONT_WEIGHT : bold
    FONT_STYLE : normal
    FONT_WIDTH : normal
    FRAMEBORDER : auto
    CONTENTSCROLLBAR : auto
    BODYSCROLLBAR : auto

Event: table
    FRAME : box
    RULES : none
    HTMLCLASS : darkbox
    CELLSPACING : 0
    CELLPADDING : 1
    FRAMEBORDER : auto
    CONTENTSCROLLBAR : auto
    BODYSCROLLBAR : auto
```

15.3 Getting the whole style

Having a separate place for style definitions is a good thing. But sometimes it is necessary to get all the style information all of the time. There is another event attribute which allows this to happen. If pure_style is set to yes, and the style attribute has been set, then all attributes of a style will show up in the event. If this behavior is desired for all events the pure_style attribute can be set as a tagset attribute instead of an event attribute. In example 15.10 pure_style is set to yes. The resulting changes to the output from the table event can be seen in figure 15.11 on the following page.

Listing 15.10: Show table style

```
proc template;

    define tagset tagsets.showstyle3;
        parent=tagsets.showstyle2;

        define event table;
           style=darkbox;
           pure_style=yes;
           trigger put_style;
        end;
    end;
run;
```

```
ods tagsets.showstyle3 style=showstyle file='style.txt';
proc print data=sashelp.class; run;
ods tagsets.showstyle close;
```

Listing 15.11: Output: Pure style

```
Event: style_class
    FRAME : box
    RULES : none
    FONT_FACE : Times
    HTMLCLASS : darkbox
    CELLSPACING : 0
    CELLPADDING : 1
    FONT_SIZE : x-large
    BACKGROUND : #FFFFFF
    FOREGROUND : #000000
    BORDERCOLOR : #000000
    FONT_WEIGHT : bold
    FONT_STYLE : normal
    FONT_WIDTH : normal
    FRAMEBORDER : auto
    CONTENTSCROLLBAR : auto
    BODYSCROLLBAR : auto

Event: style_class
    FRAME : box
    RULES : groups
    FONT_FACE : Verdana, Helvetica, sans-serif
    HTMLCLASS : table
    CELLSPACING : 1
    CELLPADDING : 3
    FONT_SIZE : medium
    BACKGROUND : #FFFFFF
    FOREGROUND : #000000
    BORDERCOLOR : #008000
    FONT_WEIGHT : bold
    FONT_STYLE : normal
    FONT_WIDTH : normal
    FRAMEBORDER : auto
    CONTENTSCROLLBAR : auto
    BODYSCROLLBAR : auto

Event: table
    FRAME : box
    RULES : none
    FONT_FACE : Times
    CELLSPACING : 0
    CELLPADDING : 1
    FONT_SIZE : x-large
    BACKGROUND : #FFFFFF
    FOREGROUND : #000000
```

```
BORDERCOLOR : #000000
FONT_WEIGHT : bold
FONT_STYLE : normal
FONT_WIDTH : normal
FRAMEBORDER : auto
CONTENTSCROLLBAR : auto
BODYSCROLLBAR : auto
```

Notice that everything is there, except HTMLCLASS. If the name of the style is really necessary, there is an event variable called style_element which is always populated with the current ods style element name.

15.4 Summary

This chapter has covered the fundamentals of using ODS styles with tagsets. The interaction is really quite simple and direct.

Chapter 16

Tagsets with Style

The fundamentals of tagsets have been covered, now it is time do something with them. This chapter will cover examples that specifically leverage ODS styles to accomplish their tasks.

16.1 A Problem with Table Rules

The tables that ods generates are fairly basic. Most people probably never take a second look at their style. Sometimes there are special needs for the way a table is done. Most often that has to do with the rules, or lines, that outline the table. Previous to SAS 9.1 ODS styles provided four attributes to control these, frame, rules, bordercolor and borderwidth. If those four don't help then things are going to be more difficult.

Table 16.1 shows the style attributes that control table rules. The extended attributes[1] are shown in table 16.2.

All of these attributes are great, but for most things all that is needed is the basic styles. A common problem is that rules is set to Cols but what is really desired is rules on the groups and columns. That can be fixed very simply with a tagset.

16.1.1 Define the problem

A good first step is to create a style that does column rules and see how that looks. The code can be seen in figure 16.1. The HTML output is shown in figure 16.1.1 on page 141.

Listing 16.1: Columnwise table rules

```
proc template;

    define style styles.table_rules;

        style table /
          borderwidth=3
          bordercolor=black
          rules=cols
```

[1] Available experimentally in SAS 9.1

Table 16.1: Table border controls

frame	Controls the frame around the table	
	Void	No frame.
	Above	Frame on top
	Below	Frame on bottom
	HSides	Frame on the Horzontal sides
	LHS	Frame on the Left hand side.
	RHS	Frame on the Right hand side
	VSides	Frame on the Vertical sides
	Box	Frame on all sides
rules	Controls the lines between the cells and accepts these values.	
	Groups	Puts rule lines between the head, body and foot
	Rows	Rules between all rows.
	Cols	Rules between all columns.
	All	Rules everywhere.
borderwidth	The width of the borders.	
bordercolor	The color of the borders.	
borderstyle	The style of the line used for borders.	
	Dotted	Dotted lines
	Dashed	Dashed lines
	Solid	Solid lines
	Double	Double solid lines
	Groove	Grooved lines
	Ridge	Raised lines
	Inset	Inet lines
	Outset	Outset lines
	Hidden	Invisible lines

Table 16.2: Extended Table border controls

bordertopstyle
borderleftstyle
borderrightstyle
borderbottomstyle
bordertopwidth
borderleftwidth
borderrightwidth
borderbottomwidth
bordertopcolor
borderleftcolor
borderrightcolor
borderbottomcolor

```
        ;
    end;
run;

ods html file='table.html' style=table_rules;
options obs=3;
proc print data=sashelp.class; run;
ods _all_ close;
```

16.1.2 A Simple Solution

The output looks fine but what is desired is a line between the Headers and the data. A little bit of research shows that the HTML thead tag can have borders. The thead tag in the output has no attributes at all. A border on the bottom of the tables head section should do the trick. As confirmation modifying the thead tag gives the correct look if the tag looks like this.

```
<thead style="border-bottom-style: solid;">
```

16.1.3 Identify and locate the event

Identifying the proper event is easy in this case. A search for 'thead' in the html tagsets gives us the table_head event.

It is important to preserve any behavior of the event that is being redefined. But in this case there isn't much there. And the event is only defined once in the entire htmltags.tpl file. Looking at the html4 tagset shows no table_head event. The parent of html4 is htmlcss, but htmlcss doesn't have a table_head event either. Finally, looking at phtml reveals the table_head event. The event is very simple.