

A SAS® Programmer's Guide to Generating Define.xml Michael Molter, INC Research, Raleigh, NC

ABSTRACT

How would you like to be able to generate a Define.xml with just a few simple PROC PRINT statements? Not possible, right? Believe it or not, with a well thought-out metadata environment, we can make this dream come true, but it won't happen overnight.

CDISC's requirement of an XML-based metadata document introduces at least three potentially new challenges to the SAS® programmer asked to generate it. One is at least a basic understanding of XML. A second is a thorough understanding of the CDISC-specific XML structure of Define.xml. A third is the use of SAS to generate it. All three of these tasks most likely lie outside the scope of what a SAS programmer building clinical databases, tables, listings, and graphs ever needed to know. This paper addresses all three of these challenges. After a brief discussion of XML basics, we will study carefully the structure of Define.xml. With these tools in place, we'll see how the ODS Markup destination provides a powerful tool for generating the final document with the familiar PRINT procedure.

INTRODUCTION

The Clinical Data Interchange Standards Consortium (CDISC) and pharmaceutical regulatory agencies (such as the Food and Drug Administration (FDA)) are introducing several changes in the clinical trials submission process, and with them, several challenges to SAS programmers in the industry. Among such challenges is Define.xml, an XML-based submission of a study's metadata designed to tell a regulatory reviewer everything they need to know about the data being submitted. Under the umbrella of Define.xml are many sub-challenges, the collection of which can make the generation of Define.xml an intimidating task. For starters, there's the organization of metadata. Something that may have received minimal attention at most, in the past, we are now not only forced to think about it, but we have to keep track of it and organize it. We have to consider not only simple data set and variable attributes such as lengths and labels, but also those that CDISC has deemed important such as the role of variables and their expected values, documentation of computational algorithms, and many more. We need to plan a database for it and we have to consider all the different ways to collect this information.

At least a database is something that we as programmers can relate to, but other challenges that lie ahead may be completely foreign to someone who has spent a career creating data sets, tables, listings, and graphs. The most obvious of these challenges is XML – a markup language that is deceitfully similar to HTML from a distance, but up close is different – a markup language that requires each of its documents to be accompanied by XML-based support documents like schemas and stylesheets. Not only is a good general base knowledge of XML helpful, but a thorough knowledge of the XML structure that CDISC has developed for submitting metadata is critical. Finally, even in our familiar, comfortable SAS environment, the challenge of writing a program that takes the metadata as its input and produces CDISC XML as its output is one to be taken seriously.

When I spell it out like that, the task is an intimidating one, and reading this paper is not going to have you turning out Define.xmls tomorrow. Companies will need to invest resources in reading several other papers written on the subject, attending talks at conferences, networking, visiting websites, and programming development time. What this paper will do is help you get your Define.xml off the ground by addressing some of these key challenges. This paper will not present a full lesson in XML, not only because of space considerations, but also because you don't need a full XML lesson. I will briefly discuss the role of the XML schema and the XSLT stylesheet, but will not discuss technical aspects of these support documents because you don't need it – at least not initially. We will look in detail at the XML structure required for metadata submission and pay particular attention to how physically separate parts of the document are to be linked. Finally, we'll look at ODS's Markup destination and ODS tagsets as tools for generating the required XML structure. Small samples of instructional SAS code will be scattered throughout this last section. In the end, the goal of this paper is not to provide you with code to copy that will automatically turn your SAS metadata into a CDISC-compliant, XML metadata document, but rather, to provide you a foundation that has helped me in my work, and to significantly sharpen your research and development focus as you begin this journey.

XML

Allow me to begin with some good news - the learning curve for the general XML knowledge needed to generate Define is not nearly as bad as you might think. Prior knowledge of HTML and markup in general will only serve to help. When you think about learning HTML, or for that matter, any other computer language, you think of two aspects. You have to learn what the keywords are, or the words that have special meaning. The second aspect is syntax, or the ways that keywords are legally combined with literal text and special characters. XML syntax is very similar to that of HTML, and so if you know HTML syntax, you're most of the way there. If you don't, there aren't very many rules to remember, and so it shouldn't take you long to pick them up. HTML has pre-defined keywords. By pre-defined, I mean that keywords such as "td", when correctly combined with HTML syntax, <td>, have meaning to

programs that read these files. Web browsers are programs that not only read the files, but display the files according to the keywords. This gives us the image of a web page. Part of learning HTML is, as mentioned earlier, learning these keywords and understanding how they contribute to what we see on a web page. As odd as it sounds, *XML has no pre-defined keywords*. In theory, users make up keywords; in our world, CDISC has done it for us. The next natural question is how made up keywords can have any meaning to anybody. This question will be answered later in this section.

So the task of learning general XML has been significantly reduced from something comparable to learning HTML. The syntax rules are few and simple, and to some, familiar. There are no keywords. And if it's all still too abstract for you, CDISC's Case Report Tabulation Data Definitions Specification (DDS) has thorough examples that give you something to aim for – a structure to which to fit your metadata. Throughout the remainder of this section we will go through the syntax rules you need to know. We will wrap it up by discussing at a high level how keywords are given meaning. That discussion will be low priority while trying to get your Define off the ground, but may become more important as you try and customize later on. This small amount of relevant theory will then lead us naturally into the next section, where we delve into CDISC-defined structure and keywords.

TAGS AND ATTRIBUTES AND CONTENT, OH MY!

Like other markups, XML is driven by elements, and each element is made up of a pair of tags, and optionally, attributes, text, and other elements or nested elements. An element begins with a start tag which consists of an open angle bracket (<) followed by a keyword that represents the name of the tag (often called the name of the element). Since CDISC has defined keywords for us, I won't go into detail about naming rules, except that they are similar to naming rules for SAS data sets - they start with letters, but can contain numerals and other characters, and cannot contain spaces. Tag names are case-sensitive, so pay careful attention to Define tags that use mixed case (e.g. <ItemGroupDef>, <CodeList>). Tags may or may not be defined with one or more attributes. When they are, the tag name is followed by a name-value pair where the "name" is a keyword representing the name of an attribute and "value" is the value of that attribute, enclosed in quotation marks, single or double. Following all attribute specifications is a closed angle bracket (>) that ends the opening of the element or the start tag.

Example 1: a start tag without attributes

```
<GlobalVariables>
```

Example 2: a start tag with attributes

```
<CodeListItem CodedValue = "Severe">
```

All start tags must eventually be accompanied by an end tag. The end tag is also constructed with angle brackets and a keyword that matches the keyword of its corresponding start tag, but two differences exist. First, whereas start tags *can* contain attributes, end tags *cannot*. Second, end tags are always preceded by a forward slash (/). What comes between the start and end tags can vary.

Example 3: a start and end tag pair in which the start tag contains no attributes

```
<Decode> ... </Decode>
```

Occasionally, start tags are followed *immediately* by their corresponding end tags. Such elements are referred to as *empty*. While this is legal, it's not always very practical, and if the start tag contains no attributes, it isn't at all practical. Either way, in the case where nothing appears between the start and end tags, XML allows us to take a shortcut. Rather than having a pair of distinct tags, we have one tag with a slash at the end. Example 4 illustrates a specific example of this kind of tag found in Define.xml.

Example 4: Short cut for an element with nothing between its start and end tags

```
<ItemRef ItemOID = "SC.SCTESTCD.ALLERGY" OrderNumber = "1" Mandatory = "No"/>
```

Alternatively, between its start and end tags, an element can contain *content*. Content can be in the form of more elements (nested elements), text which contains no tags, or both – also referred to as mixed content. More elements are sometimes described in relation to the element in which they are nested, such as subelements, children, grandchildren, etc, and the element within which they are nested is referred to as a parent element. These subelements follow the same rules that parent elements follow - naming conventions, start tags with or without attributes and end tags, each surrounded by angle brackets. After the end of a start tag, if the next character is not an open angle bracket, then what follows the start tag is text. Text has no special characters to distinguish it and no rules governing its content. We sometimes refer to this text as the value of the element. This value continues until the end tag of the enclosing element. Example 5 illustrates both subelements and content. The TranslatedText element, a subelement of the Decode element, has the value "NO".

Example 5: Subelements and text

```
<Decode>  
  <TranslatedText xml:lang = "en">NO</TranslatedText>  
</Decode>
```

XML STRUCTURE

We're now ready to take a step back from the details of the syntax and look at the broader structure. Though general XML doesn't require it, many XML documents including Define.xml begin with an XML declaration. Define's is illustrated below.

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

Except for the declaration, XML elements in which a question mark immediately follows "<" and immediately precedes ">" are referred to as XML processing instructions. Rather than showing up in a browser, these pass along instructions to an application that reads the XML. In Define, CDISC allows us to follow the declaration with a stylesheet processing instruction, or an instruction that tells the browser to apply a stylesheet before showing it to the user. We'll discuss stylesheets a little more in the next section, but for now, we illustrate Define's optional stylesheet instruction below.

```
<?xml-stylesheet type = "text/xsl" href = "define1-0-0.xsl"?>
```

The value of the HREF attribute is the name of the external stylesheet.

Technically, the declaration and the stylesheet instruction are not considered XML elements. They do use the angle brackets as delimiters like tags, but note that unlike elements, they do not come in start-end pairs or have the slash at the end.

Immediately after these, every XML file must have the start tag of what is known as the root element. The end tag of the root element comes at the end of the file. Put another way, the root element has no parent or sibling elements, and every other element is a descendant of it or nested within it. Define's root element is the ODM element. Within the root element are properly nested subelements. We saw earlier that following the start tag of an element can be text or the start tag of a subelement. "Properly nested" means that any element must close (or in other words, its end tag must appear) prior to the closing of its parent element. In other words, child elements must be opened and closed entirely within their parent elements (or between the start and end tags of their parent elements).

Example 6: Proper nesting

```
<GlobalVariables>
  <StudyName>1234</StudyName>
  <StudyDescription>1234 Data Definition</StudyDescription>
  <ProtocolName>1234</ProtocolName>
</GlobalVariables>
```

Example 7: Improper nesting

```
<GlobalVariables>
  <StudyName>1234</StudyName>
  <StudyDescription>1234 Data Definition</StudyDescription>
  <ProtocolName>1234
</GlobalVariables>
</ProtocolName>
```

XML SCHEMAS AND XSLT - GIVING MEANING TO YOUR XML

So what does it all mean? How can XML have any meaning if I'm allowed to make up element names and attributes? To answer that question, let's go back in time and consider the early days of HTML development. At some point, someone made up elements like <table>, <tr>, <td>, and so on. Today we know these as elements that begin the display of a table, the display of a row in a table, and the display of cell content in a table, respectively, but in the beginning, they were as made up as <GlobalVariables> and <StudyName> illustrated above. They took on their roles as table display elements when someone got around to writing an application – a web browser – that parsed the HTML text and associated elements like these and others with display characteristics. In beginning-level XML texts where it's common to draw comparisons between XML and HTML, you'll see that HTML is markup that's concerned with the display of data.

In the same paragraph in that text, you'll also read that XML isn't concerned so much about the display of data as it is about the transporting of data. XML can be transported easily across machines because it's simple text that any machine can read. HTML is simple text that applications we call web browsers can read. These applications not only read it, but they do something with it. Specifically, they separate or *parse* the elements and attributes and execute display instructions as they process them. So if XML and HTML are nothing more than text, then if we can write applications that read HTML text and do something with it, then we should be able to do the same thing with XML text. In particular, in collaboration with CDISC, regulatory agencies such as the FDA have written applications that read XML such as that found in Define and load the data into data warehouses. A second example is SAS's XML engine that acts as an application that reads XML and transforms it into SAS's proprietary data set format. Internet

Explorer can also be considered an XML application, although all it does is display the markup in collapsible sections that are defined by the elements.

VALIDATION

XML applications are written with expectations. For starters, as input, they expect syntactically correct or *well-formed* XML. Additionally, in order to “do something” with XML content, the application has to be able to identify the content, and it does that by referring to element and attribute names. So what does an XML application do when these expectations aren't met – when the syntax is wrong, or when elements identified by the application aren't present?

We can begin to understand the answer to this by asking ourselves the same question about a simple SAS program. Consider a program that begins with a DATA step. Before it begins executing the statements, it goes through a compile stage, where it checks for syntax; it checks to see that the data set specified by DATA= exists, and it checks for several other things. If everything is ok, it will execute the statements, but if any fail it will issue log statements and not even try to execute. And where do these checks come from? They're built into the SAS system or the software application.

In some respects XML validation is similar. A computer language's syntax is what makes it useful, readable. Without it, it's just meaningless text. When our SAS syntax is incorrect, the SAS system has no way of knowing how to execute. Similarly, we can't expect XML parsers to read XML that is not well-formed. In a sense, we could say that XML that is not well-formed isn't even XML. For that reason, just like at SAS's compile time, when we ask an application to process XML, the parser first performs a syntax check. Violations are reported and prevent the application from performing its intended function.

While “well-formed” refers to validation on a syntax level, the term *valid XML* or *semantic validation* usually refers to another level of validation. Just as data sets named in a SAS program are expected to exist, so too are elements, attributes, and content referred to in XML applications. While this kind of validation is built into the SAS system, an XML application's validation at this level is often defined in a separate XML file called an XML schema. An XML schema is a well-formed XML document with pre-defined element names and attributes that lays out the rules for any XML document that chooses to use it. Schemas have the power to specify which elements and attributes are to be used, what kind of data (string, datetime, integer, etc.) an element contains, how many instances of each element is allowed, the order of the elements, and more. Because it's a separate document, enforcing such rules for a particular XML document (the *instance* document) requires a reference to the schema. It's then up to the application to read this reference, use the schema to validate the instance document and act on its findings. How it acts on these findings is up to the application and how it's written. We know that SAS does the best it can to continue processing subsequent parts of a program even when an earlier part failed. Of course this fact has its own ramifications. If the MEANS procedure fails and a subsequent DATA step attempts to read the data set output from PROC MEANS, then that DATA step will most likely fail too. Some XML applications may be written to quit after one violation of the schema is found. Others may try to move on and report all findings. Finally, others may not depend on a strict structure and may ignore certain kinds of schema violations or not use a schema at all.

CDISC has created an XML schema document, define1-0-0.xsd, that we must reference as an attribute of the ODM element. This reference is found as two attributes of the ODM root element and is illustrated below.

```
<ODM
  xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemalocation = "http://www.cdisc.org/ns/odm/v1.2 define1-0-0.xsd"
```

Internet Explorer is one application that can do a syntax validation of our Define. When an XML instance is not well formed a web page appears with the message “The XML page cannot be displayed” and a description of the error. Unfortunately, IE cannot do a semantic validation of Define, but other validation programs may be available on the internet.

VIEWING DEFINE.XML

Up to this point the meaning we've attached to an XML document's homegrown keywords has been discussed only in terms of our ability to extract and store XML content. Validation is often built into an application to check an XML document's “extractability”. What we haven't yet touched on is our ability to view an XML document. In fact we stated earlier that XML isn't concerned about display, but that doesn't mean we don't want to see it. CROs and sponsors may wish to use a more readable version as a reference guide for their data. For that reason, we briefly discuss here the XSLT stylesheet.

XSLT, an acronym for XML Stylesheet Language Transformation, is an XML-based transformation language that has the ability to “transform” your markup into something viewable like HTML. We mentioned earlier that opening Define.xml with Internet Explorer simply displays the markup in color-coded collapsible sections. While the ability to collapse and expand elements helps to eliminate clutter when looking at markup, markup transformed into tabular format is much easier to process cognitively. XSLT transforms XML to HTML, or put another way, generates HTML

in a way similar to the way the macro facility generates SAS code. Both can generate literal text conditionally or unconditionally, with or without the aid of similar kinds of programming logic. Where they differ is to what conditional and programming logic is applied. Macros often use as input user-provided parameters, whereas XSLT uses XML content such as data and attribute values as its input. In order to do that, the XSLT language uses XPATH, a language used to find and extract information in an XML document. Consider Example 8 below.

Example 8: an excerpt of an XSLT stylesheet

```
<table border='2' cellspacing='0' cellpadding='4' bgcolor = "#fffd da">
  <tr> <th colspan='7' align='left' valign='top' height='20' bgcolor = "#ECECEC">
Data sets for Study

<xsl:value-of select = "/odm:ODM/odm:Study/odm:GlobalVariables/odm:StudyName"/>

</th> </tr>
```

Note that everything through the text "Data sets for Study" is valid HTML markup, and is being generated unconditionally, without logic, by the stylesheet. What is to follow the word "Study" is the name of the study, which is information to be found in Define.xml. This information is extracted using the xsl:value-of element and the select= attribute. The value of the select attribute is an XPATH expression that tells the stylesheet which branch of the XML tree to navigate through to extract the text.

Unlike a schema, stylesheet references are not required by XML or by Define.xml, though some sponsors may require it for their own purposes. Because of that, CDISC does not currently take responsibility for maintaining such files, though two different versions are available on their website (one is for members only).

TO SUMMARIZE...

What does this all mean to you? Up to this point we've covered a mix of theory and practice - Define.xml practice. Before getting into the details of Define.xml in the next section, let's summarize among the theory we've covered what is particularly relevant to you as you get your Define off the ground.

We began by comparing XML education to HTML education and quickly saw that for our purposes, learning XML isn't nearly the chore it might appear to be. In part this is because the task of learning pre-defined keywords that is common to understanding so many other computer languages does not exist for understanding general XML, although understanding that fact does take some getting used to. We also discussed syntax. As we'll see, we can learn a lot from the Define samples that CDISC makes available for us, but by taking for granted the proper syntax that these samples illustrate, we lose the ability to comprehend messages that applications deliver to us when we do violate these rules. For this reason the section above regarding the basic syntax rules such as proper nesting, enclosing attribute values in quotation marks, etc is worth studying and becoming comfortable with.

Believe it or not, the thoroughness of examples such as those found in the DDS combined with some basic knowledge of syntax goes a long way toward at least knowing what metadata goes where in Define. In the next section we'll use the DDS examples to discuss Define keywords and linking relationships built into Define, and that will take us even further. A second tier of learning, defined as concepts you may not need right away but would be beneficial to study along the way, would include schemas, stylesheets, and namespaces. We didn't discuss namespaces and CDISC tells you through its examples how to declare and reference them, but knowing something about them will explain the xmlns: attributes in the root element, as well as the special meaning of elements and attributes that are prefixed with colons. Because CDISC has created a schema for you, you don't have to create your own, but understanding the schema language and how to read define1-0-0.xsd can help you understand structural requirements not illustrated in examples. Using stylesheets published by CDISC may be enough, but an understanding of XSLT allows you to customize for your customers (e.g. hard-coded text, colors, column/section inclusion/exclusion), and in some cases, troubleshoot issues such as hyperlinks not working, which may lead to resolution of Define violations.

THE XML OF DEFINE.XML

At this point, we know a few XML specifics; we know of the existence of other things whose details we need not be concerned with, and we know almost nothing about Define.xml. We know that XML syntax, though illustrated well in published examples, is important to learn right away. We know about semantic validation through an XML schema, but nothing about schema language; about viewing an XML file through a web browser with the help of an XSLT stylesheet, but nothing about the XSLT language. While not top priority, these should certainly be somewhere in the middle of your to-do list. Finally, we know that while general XML does not come with a pre-defined set of tags like HTML does, CDISC has developed a tag vocabulary for us. It's now time for us to get to know not only the vocabulary, but also how the elements are to relate to each other.

We'll begin this section by identifying some valuable resources you should have nearby as you get started. Using that, we'll go through each section of the markup. As we do this we'll discuss the general purpose of each section, but we'll leave out details that are available in other resources. We will then conclude by discussing how different sections are related to each through common attribute values.

SOURCES

As you might expect, the source of most, if not all of your resources will come from CDISC's website - www.cdisc.org. Here you'll find a Standards link which takes you to a list of the standards, split between those in production and those in development. Among those in production is the Case Report Tabulation Data Definition Specification (`define.xml`). Details on this standard can be found by clicking on the hyperlinked text, (CRT-DDS V1.0), found alongside of this description. Along with a detailed description of the standard, you will find links to several files available for viewing or download. These include schema and stylesheet files, Define examples, and a PDF file called `CRT_DDSpecification1_0_0.pdf`. It is from this 45-page file (DDS) that we will be working throughout most of the remainder of this section.

After a few introductory pages, most of the remainder of the DDS is divided into the sections of Define. In each section is an introductory paragraph about its purpose. Following that are tables that describe the section's XML elements and attributes including whether or not they're required. Following the tables are snippets of examples and illustrations of how they are related to other parts of the Define document. After this, beginning on page 39 of the DDS and ending on page 43 is a sample Define, and on page 44 is an outline of the elements that illustrates nesting relationships.

One other important source of information is worth mentioning here. From the CDISC.org home page is another link labeled Public Discussion Forums. While the DDS is thorough in its explanations and its examples, there are always questions it doesn't answer and situations not illustrated in the examples. As of this writing the DDS is also old and due for revisions. The discussion forums are exactly what they sound like - users from around the world of all levels of experience asking questions. Answers come from other users, as well as members of the CDISC team. Together the forums cover several areas of CDISC, one of which is dedicated to `Define.xml`. The reader is encouraged to click on this link, briefly set up a profile (or just log on as a guest), and simply browse over the history of questions and answers. A profile is required in order to participate in discussions.

UNDERSTANDING DEFINE.XML

In order to understand the pieces of `Define.xml`, we begin with its structure. In layman's terms, we can think of `Define.xml` and its sections as a book about our study. A book often begins with some combination of a forward, a preface, and acknowledgments. Following that is a table of contents, the body of the book (its chapters), and at the end, appendices that might include an index or explanation of endnotes. In a similar way, the DDS describes Define in terms of a table of contents and Data Definition tables, but we can take it a step further. Define contains a header and high-level information about the study to correspond to a book's forward. Define's table of contents, illustrated in tabular form on Page 6 of the DDS, consists of "chapter names" that each correspond to a data set in the database, accompanied by data set attributes such as labels, structure, keys, and more. Define contains an element structure to support this in the `ItemGroupDef` element. The body of the book is then made up of chapters or Data Definition Tables, each of which corresponds to an item in the table of contents, and provides detailed metadata for the corresponding data set - namely, attributes of each variable in the data set. A sample chapter is illustrated on page 10 of the DDS in tabular form. Define contains this structure in the `ItemRef` element, a subelement of `ItemGroupDef`, and the `ItemDef` element. Finally, Define has its own version of endnote appendices. Just as many terms and phrases in a book require further explanation in an appendix because they're too long to include in the body, many variables in a database require explanation beyond what can fit comfortably into the Comments column. Such information includes explanation of computational algorithms, lists of possible values and decodes (controlled terminology), and explanations of the mixed nature of certain variable values (value-level metadata).

This analogy is easily illustrated when viewing a Define that includes a stylesheet reference to `define1-0-0.xsl`, CDISC's published stylesheet, with Internet Explorer. The table of contents appears at the top of the web page and looks like the example on page 6. All of the data definitions tables appear after the table of contents, and are also accessible with hyperlinks from the data set name column of the table of contents. The appendices or endnotes follow, which are accessible with hyperlinks from associated variables in the data definitions tables.

From our discussion so far, we can see that Define contains a web of connections or links between different parts of the document. At first this web may seem like something that exists only in theory without any manifestation in the markup, but when we consider the fact that the stylesheet is able provide access to one end of the web from another through hyperlinks, we're forced to think otherwise. It is these relationships that will be the main focus of the remainder of this section. While some attention will be given to element structure, the thoroughness of the examples provided in the DDS as well as the global element order on page 44 and the tables that explain the meaning of each element and attribute make it unnecessary to repeat them here. Since the DDS makes clear the order in which elements are to appear, we will continue with the book analogy, noting how relationships are manifested in the markup. We'll also note places where this order is slightly different from the element order. In the end, beyond

element and content meaning and order, or beyond just the words printed on the pages, we will have seen the story that Define is trying to tell.

We begin our story with the forward - an idea of what is to follow in the book. Define's counterpart includes the XML header plus high-level study information. In our earlier discussion of general XML we illustrated concepts with examples from Define.xml. Putting those examples together, we get example 9 illustrated below, and also found on page 26 of the DDS.

Example 9: a sample XML header

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<?xml-stylesheet type="text/xsl" href="define1-0-0.xsl"?>
<ODM
  xmlns="http://www.cdisc.org/ns/odm/v1.2"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-Instance"
  xmlns:xlink="http://www.w3.org/1999/xlink"
  xmlns:def="http://www.cdisc.org/ns/def/v1.0"
  xsi:schemaLocation="http://www.cdisc.org/ns/odm/v1.2 define1-0-0.xsd"
  FileOID="Study1234"
  FileType="Snapshot"
  ODMVersion="1.2"
  CreationDateTime="2008-12-02T13:39:06">
```

We know that the first line is the XML declaration and the second is the stylesheet reference. The third line opens the ODM root element, which is then followed by several attributes before the start tag is closed. Those that begin with xmlns are declaring namespaces and those with a colon following xmlns are associating prefixes with the namespaces. The one without the colon is the default namespace. While the name of the stylesheet could change from one Define to the next (or may not even be referenced in all Defines), the root element and the namespace declarations should appear in your Define exactly the way they appear here - they are not expected to change from one Define to another. The ODMVersion attribute will also remain the same until Define is incorporated in version 1.3.

As seen on page 44 of the DDS, the only first-generation child of the ODM root element is the Study element. Because Study has no siblings, its end tag immediately precedes the ODM end tag. Study is then partitioned by two child elements - GlobalVariables and MetaDataVersion. While the three children of GlobalVariables provide more high-level study metadata, children of MetaDataVersion begin with information about external documentation, and then move into more detailed metadata. It is with the external documentation elements - def:AnnotatedCRF and def:SupplementalDoc - that we begin to see linking between separate parts of the markup.

Sponsors are allowed to submit any number of annotated CRFs or any other supplemental files that they think may aid a reviewer. def:AnnotatedCRF and def:SupplementalDoc are elements that each contain one or more def:DocumentRef child elements. def:DocumentRef is an empty element which contains only one attribute – an identifier attribute called leafID. Throughout this paper the term “identifier attribute” refers to an XML attribute whose text string value has nothing to do with any metadata or even the study itself, but is used to match the value of an attribute in another element of the document, thereby creating a link or a relationship between the two elements. Such attributes appear frequently throughout Define. It's similar to a SAS data set variable that's used in a BY statement to merge with another data set, but that has no purpose beyond this. In the current situation, def:AnnotatedCRF contains one def:DocumentRef for each annotated CRF file, and def:SupplementalDoc contains one def:DocumentRef for every other linked external file. The leafID attribute of each of these DocumentRef elements contains an arbitrary text string value that matches the value of the ID attribute (another identifier attribute) in a def:leaf element - a sibling to def:AnnotatedCRF and def:SupplementalDoc. In addition to ID, def:leaf contains the xlink:href attribute, whose value contains the relative path (relative to the directory in which Define is located) of the external file. The description of this file or the text that contains the hyperlink is contained in the value of the def:title element - a necessary child element of each def:leaf element.

One use of the AnnotatedCRF element comes from Define1-0-0.xsl. The stylesheet scans through the list of leafID attributes found in the def:DocumentRef elements, and for each one it finds, it uses the leafID value to go find its corresponding path and description information. Once found, it then creates a hyperlink to it. Once finished with the AnnotatedCRF list, it does the same with the SupplementalDoc list.

Like the forward in a book, Define's forward may not play much of a role in the story, but it is still necessary. Though the author's family isn't part of the story, the author often states in the acknowledgments that they couldn't have written the book without their support. Similarly, XML header information isn't found in the tables seen in the readable Define, but the document cannot be processed without it. The rest of the forward - the high level study information such as study name, description, and protocol - does not lend itself to tabular presentation. References to some of it can be found scattered throughout the readable version. Examples include “Data sets for study”

followed by the value of the StudyName element, links to annotated CRFs and supplemental documents, and the creation date found throughout.

We're now ready to move on to the table of contents and the data definition tables. When we think of a book's table of contents, we think of a list of chapter names, and for each, a corresponding page number where the beginning of the chapter can be found. Put another way, the chapter names in the table of contents can be thought of as high-level descriptions of chapters or pieces of the story to be found later, and the page numbers are shortcuts to them. Define's chapters are its data sets or its data definition tables. Its table of contents is a table found in the beginning of Define and contains one row per data set. Each chapter is described in terms of data set-level attributes such as labels, data set structure, purpose, keys, and the location of the corresponding SAS transport file. The chapters themselves describe each of the variables in the data set, including label, origin, comments, role, etc. Being an electronic file, page numbers in the table of contents that take you from the table of contents to any data definition table are replaced by hyperlinks. Let's now see how all of these aspects - data set descriptions, variable descriptions, and links from one to the other are manifested in the markup.

Each data set's descriptors are housed in an individual instance of an ItemGroupDef element. ItemGroupDef is a child element of MetaDataVersion, and because each data set gets its own ItemGroupDef, it is the first child of MetaDataVersion that has multiple instances (recall that def:DocumentRef could have multiple instances, but is a grandchild of MetaDataVersion). By looking at the DDS, the reader will note here that we have deviated from the element order required by Define to talk about ItemGroupDef. As you might expect, many of a data set's descriptors are kept in attribute values. For example, the Name attribute is the name of the data set, def:label holds the data set label, and so on (see page 7 of the DDS for the full list). Other important information however is kept in subelements.

The table of contents not only contains the name of SAS transport files, but it also contains hyperlinks to them. By thinking of these links as links to external files, it should then come as no surprise that markup that lies beneath these links is similar to the markup behind links to annotated CRFs and other supplemental files discussed earlier. Note that one of ItemGroupDef's attributes is def:ArchiveLocationID. Like the LeafID attribute discussed earlier, this is also an identifier attribute. Being an identifier attribute, its value is not part of metadata, but must match the value of another identifier attribute. Just as this matching attribute was found in a def:leaf element, each ItemGroupDef must contain a child def:leaf element. Recall that def:leaf has the ID attribute. In this case, the value of ID must match that of def:ArchiveLocationID. def:leaf must also contain the child def:title element whose value is the text that contains the link.

So where in the markup is the information needed to build each of the data definition tables? For any given table we *start* with information still found in ItemGroupDef. Among the descriptors of a data set is the role that each variable plays in that data set. This is manifested in the ItemRef element - another child of ItemGroupDef. Every variable in the data set has a corresponding ItemRef. ItemRef is an empty element and so has no values or children, but contains attributes such as Role, whose possible values (or controlled terminology) can be found on page 11 of the DDS, OrderNumber whose integer value describes the variable's physical location in the SAS data set, and Mandatory whose value indicates whether or not the absence of values would render the item incomplete. A data set's data definition table is constructed from this list of ItemRefs, and so contains one row for each variable described in an ItemRef. The existence of these ItemRefs within ItemGroupDef makes it easy to create a hyperlink (using the Name attribute of ItemGroupDef) from the table of contents to the data definition table. There does appear to be one problem though - if you look closely at the data definition table, only one of its columns is an attribute of ItemRef. So where in the markup are the other column values such as Origin, Label, and Comments?

Define requires us to describe a variable in many ways. As mentioned, attributes such as OrderNumber and Role found in ItemRef elements speak specifically to a variable's role *in that data set*. Other variable descriptors, however, may be more universal. In other words, a variable that might be found in multiple data sets may be described in all data sets with the same set of attribute values. Rather than forcing us to repeat the same set of values in every data set a variable is found in, Define allows us to describe these attributes outside of the context of any specific data set. In markup terms, they are described outside of the ItemGroupDef elements - in ItemDef elements.

For example, consider the STUDYID variable. For reference, the ItemDef attributes can be found on page 12 of the DDS. STUDYID is required in all data sets, and so each ItemGroupDef must contain an ItemRef for it. In theory different data sets might have different values for any of the ItemRef attributes that describe STUDYID, but if one set of attribute values for the attributes found on page 12 describe STUDYID in all data sets, then we can get away with just one ItemDef instance. On the other hand, perhaps you choose to include a comment with USUBJID in the DM data set, but not with any other data set. In this case, you will need two ItemDefs for USUBJID - one that includes the comment that will be linked to DM, and the other that doesn't, that will be linked to all other data sets that have USUBJID.

The remainder of the columns in the data definition tables comes from these ItemDefs. Since each table is made up of information from both ItemRef and ItemDef elements, the markup needs to link these elements. Once again this is done with identifier attributes. ItemRef's ItemOID attribute is an identifier attribute whose value must match the OID attribute value of its corresponding ItemDef element. In SAS terms, think of ItemRef as a data set that contains

some, but not all of the necessary information about a list of variables. To get the rest of the information, you have to merge with another data set - ItemDef. The BY variable in this merge would be ItemOID from ItemRef and OID from ItemDef (of course one of them would have to be renamed).

Let's return to the USUBJID example. If this variable is to contain a comment in the DM data definition table, and no comment in any other data definition table such as AE, then we would need one ItemDef that would have the comment, and another that didn't. Additionally, the value of the OID attribute in the first ItemDef would match the value of the ItemOID attribute in the ItemRef element that describes USUBJID within the ItemGroupDef element for DM. Similarly, the OID from the second ItemDef (without the comment) would match the ItemOID in the ItemRef that describes USUBJID within every other ItemGroupDef that contains USUBJID. The relevant markup of such a situation is illustrated below. Additional examples are found on pages 15 and 16 of the DDS.

Example 10: ItemDef - ItemRef relationship

```
<ItemGroupDef name="DM" <ItemRef ItemOID="DMUSUBJID"> >
<ItemGroupDef name="AE" <ItemRef ItemOID="USUBJID"> >

<ItemDef OID="DMUSUBJID" Comment="DM comment">
<ItemDef OID="USUBJID">
```

APPENDICES

While reading through a book we often find words that are immediately followed by superscripts. We know that this indicates that the preceding word(s) require explanation whose length might interrupt the flow of the story if inserted in the current location. For that reason these explanations are usually placed at the bottom of the page or in an appendix at the end of the book, where the reader can read them when ready.

Many of the variables in the data definition tables also require explanation. Of course the data definition table has the Comments column, but there are two circumstances under which an appendix at the end might be better suited than the Comments column to provide the explanation. Similar to the book analogy, length is one of those reasons. Long lists of controlled terminology squeezed into one cell would make the row for that variable very tall. A second reason is one we've seen before, when we discovered the convenience of one ItemDef instance for documenting the same attributes for a variable found in several data sets. While an explanation for a variable may be easily short enough to fit into a column of a data definitions table, the same explanation may explain several variables throughout the database. Rather than repeating that explanation, we document it once in the appendix, and all the variables to which it applies can reference it. Like the table of contents, the electronic counterpart of page turning is hyperlinks, and as we've seen before, this suggests the need for identifier attributes in the markup to establish these links.

Before we get into the markup, let's briefly look at the three different types of explanations for which markup is available in Define. One is a computational algorithm. Some derived variables should be accompanied by an explanation of how they were derived. A second is controlled terminology. Certain variables have discrete lists of possible values and CDISC requires that they be documented. When these values are coded, we must also provide the decode. A third type which is a little more complicated is value-level metadata. CDISC requires that SDTM Findings data be submitted in a normal or vertical data set structure, even when multiple tests are involved. This means having a variable whose values indicate the different tests (--TESTCD and --TEST) and a results variable (--ORRES). Because each of these tests (each of which can, conceptually, be thought of as a variable of its own, and are sometimes delivered that way by the DBMS) can theoretically have its own set of attributes and its own controlled terminology, each needs to have its own metadata documented. This is done in the value-level appendix.

Because the existence of further explanation for a variable can be thought of as an extension of its own metadata, the markup link begins in all three cases within ItemDef. In the case of the computational algorithm, def:ComputationMethodOID is an identifier attribute of ItemDef (illustrated on page 24 of the DDS). In the case of the controlled terminology, the empty CodeListRef element appears as a child to the ItemDef, with an identifier attribute called CodeListOID (page 19-21). Similarly, def:ValueListRef is an empty child of ItemDef with ValueListOID as its identifier attribute. This is illustrated on page 23, but note that that illustration refers to the identifier attribute as def:ValueListOID, whereas the example on page 42 leaves off the def: prefix. Technically both are correct, but Define1-0-0.xsl expects no prefix. Links won't work if the prefix is included.

We know that the hyperlink starts in the data definitions tables and ends in an appendix table. Naturally, hyperlinks that start in the data definitions table have corresponding markup in ItemDef elements as we've just seen, but we now consider the markup that corresponds to the target of these hyperlinks. For computational algorithms and controlled terminology, it's straightforward. The def:ComputationMethod element is another child element of MetaDataVersion that appears before the first ItemGroupDef element. The structure of this element is simple: an identifier attribute called OID whose value matches that of the def:ComputationMethodOID attribute in the ItemDef. The description of the algorithm is then the value of the element. This is illustrated on page 24 of the DDS. Similarly, the value of CodeListOID matches the value of the identifier OID attribute in the CodeList element - another child of MetaDataVersion that appears toward the end of the markup. CodeList is a more complex element with more

attributes, plus one instance of a child CodeListItem element for each discrete value of the controlled terminology. CodedValue is an attribute of CodeListItem, and the decode is buried as a value of the TranslatedText grandchild element. The Decode element is between CodeListItem and TranslatedText. More details can be found in the DDS.

Value level metadata is more complicated because for any variable subject to it (e.g. --TESTCD), each of the possible values of the variable must also be treated like variables. The --TESTCD variable itself is similar to the other explanation types - the value of the ValueListOID attribute must match the value of the identifier OID attribute found in another child of MetaDataVersion - def:ValueListDef - also found before the ItemGroupDef elements. Since each of this variable's values is also being treated as a variable, we also need an ItemDef for each of them. Keep in mind that most ItemDefs are linked to an ItemRef within an ItemGroupDef. Since these are not actually variables in a data set, ItemRef information such as OrderNumber, Role, and Mandatory can't be found in any ItemGroupDef. Rather, they are linked to ItemRefs that are children of the def:ValueListDef attribute just mentioned. This relationship is well illustrated on page 23 of the DDS.

CREATING DEFINE.XML WITH SAS

Until now we've discussed *what* needs to be produced; it's now time to talk about *how* to produce it with SAS. In particular, in this paper we will discuss the use of the Output Delivery System (ODS). If you're a frequent user of ODS this might come as some relief to you thinking that it's as simple as a couple of ODS statements that open and close destinations and maybe a few ODS options, but two questions arise that must be addressed whose answers are not simple: which ODS destination will we use, and what data sets hold the metadata to be output to this destination?

To the first question, the short answer is the markup destination - but that's only the beginning. Opened without any options, this destination produces valid XML with a generic set of tags and attributes. Though valid, it is certainly not CDISC xml, and as of the writing of this paper, SAS has no destination that will produce CDISC xml. For that reason we have to create our own destination with an ODS *tagset*. Though a full explanation of how to create tagsets and all the tools available to do so is beyond the scope of this paper, a significant portion of this section will be dedicated to discussing important concepts of output delivery with tagsets, including some of the PROC TEMPLATE syntax, the event model, and tagset variables. We'll also spend some time talking about capturing system information and PROC results and writing it all out to the output file. For a more comprehensive discussion, the reader is encouraged to read any of the selections mentioned in the references section at the end of this paper. In the end, though a complete tagset will not be provided, adequate discussion of the issues to consider should be enough to get the reader started on building their own.

Of course just mentioning the Output Delivery System as a tool for generating output implies that we have something to output - most likely, something that started in one or more SAS data sets. In our case, using ODS means that we have to organize our metadata into one or more data sets, or what we'll call a metadata database (MDDB). In reality, your metadata will come from one of many different sources. Some of it, such as variable attributes, may be captured programmatically, while others may be hard-coded from protocol text. In this paper all discussions of using ODS will be based on the assumption that the reader has decided on a structure for their MDDB and a method for building it that will meet their needs. Some may decide to use Excel spreadsheets for data entry while others may use SAS's PROC FSEDIT; some may create one data set for each domain in their MDDB while others may choose to combine the metadata from all domains into one data set; some may choose to put all controlled terminology into one data set while others may split it into multiple data sets. Although examples will assume specific structures, they are for demonstration only and are not meant to suggest best practices.

After general tagset discussion we will immediately dive into capturing and writing PROC PRINT results. In the end, the object will be to have a tagset to be used with the markup destination that will allow us to simply apply PROC PRINT to all data sets in the MDDB in the proper order once the destination is opened, as in the example below.

Example 11: Generating Define.xml with ODS's markup destination

```
ODS markup tagset = your-tagset-name file = "filename.xml" ;
proc print noobs data = MDDB-data set-1 ; run;
proc print noobs data = MDDB-data set-2 ; run;
proc print noobs data = MDDB-data set-3 ; run;
etc.
ODS markup close ;
```

THE MARKUP DESTINATION

When we think of ODS, we think of the different file formats we can send our results to, such as PDF, CSV, RTF, and HTML. We do this with a small handful of statements with a handful of options, and *POOF* - we have our output. With trivial changes, we can easily change formats. We do this without giving hardly any thought to the work that ODS has to do - capture the results of the PROC and surround them with just the right markup in just the right places.

System and PROC options, global statements such as TITLE and FOOTNOTE as well as global ODS statements, and the ability to manipulate style and table templates gave us more freedom to customize output than we ever thought we needed. Then came XML.

Users realized that while the options, statements, and templates that they could manipulate gave them indirect access to certain parts of the markup, other parts of the markup were untouchable. This realization became most apparent when SAS programmers began to have a need for generating XML. RTF and HTML were one thing - they had pre-defined keywords (tag names, attributes, etc) that SAS could hard-code into the software - but how could SAS be expected to create markup like XML that had no pre-defined tags, a markup whose tag names we get to make up? How could SAS be expected to generate text such as `<ItemGroupDef` or any other text that CDISC has defined? SAS does have an XML destination but its keywords and output structure are not what is required by CDISC. Users now need more than just statements and options for ODS to translate into markup - they need direct access to the markup. They need the ability to bypass the translation and write the markup themselves. That's where tagsets come in.

Tagsets have been around in one form or another since ODS has been around. After all, it takes more than magic to start with an ODS statement that opens a destination and another to close it, and end with complex markup. Somewhere in SAS's source code has to be instructions on how to create that markup. In addition to these instructions, the source code also has to have the ability to capture dynamic information. This includes not only the results of PROCs, but also text from TITLE and FOOTNOTE statements and any other global or system information that needs to be translated into markup. In a nutshell, when SAS gave users tagsets, they handed over this source code.

On the surface, the idea is a simple and familiar one. After all, since long before ODS has been around we've known how to use a DATA step to capture and manipulate data, and write a combination of hard-coded literal text and manipulated dynamic data to a text file one record at a time. The tagset, written with the TEMPLATE procedure, is the same set of instructions applied to PROC output (and other global parameter values such as titles and system options). We'll refer to this as markup data. The tagset has many of the data manipulation tools that the DATA step has including most of the same functions, variable and array functionality, statement blocks, DO/WHILE iterative looping, conditional logic, and more, though much of the syntax is somewhat different. Of course the FILE statement is unnecessary because the output file is named in the ODS statement that opens the Markup destination, but it does have the PUT statement (and some derivatives of PUT) with most of the same functionality (line pointers and trailing @s are among the DATA step PUT features not available in the tagset's PUT). In other words, once data is available, a tagset can manipulate and write it out the same way the DATA step does with minor differences in syntax. What's different is how data is captured.

We know that data is available to the DATA step through variables and each carries different information about each record. Markup data is also available through variables, but it doesn't come from data records. We also know the DATA step as an *active* piece of SAS code - every executable statement is sequentially applied to every record read. A tagset is a template, and template code is more *passive*. It's normally separated into pockets of code, each of which is only executed when it's called upon from somewhere else. In an ODS style definition, these pockets are called elements, each of which is executed when it's loaded into a stylesheet. In a tagset, the pockets are called events, and an event's code is executed when ODS is building a file and calls for that event.

That's how ODS builds a file - in a sequence of discrete steps, at each of which it executes a specific event's code defined in the tagset. This is how ODS has always worked. In the early days, tagsets and their event definitions were simply part of SAS's source code. Today, SAS makes its tagsets available to us, and through PROC TEMPLATE, we can inherit and modify them. Additionally, we can use PROC TEMPLATE to create our own from scratch. The latter is what we need to generate Define.

So how does ODS know which events to call? The answer depends on what part of the file it's building, but for the most part, the pattern of event calling is consistent with the markup pattern found in markup files that contains tables. Most files begin with a high-level header section that consists of "one-time" elements. Correspondingly, in the beginning, ODS calls events, each of which should be defined to write markup normally found in the beginning, and each of which should only be called this one time. Some tagsets may be defined to allow for embedded stylesheets. In such cases, ODS will call the same event over and over again, in each case, with differences in *style variables*. These style variables get their values from the ODS style definition being used. Each call corresponds to a different element in the style definition, and results in a different style class in the embedded stylesheet.

These first few events, all called before the PROC is processed, are fairly static, except for the stylesheet events which are only called when a stylesheet is to be included, and whose variable values are derived from the style definition specified on the `style=` option on the ODS statement that opens the destination. In fact since these are called before the PROC is processed, you can close the destination without executing any PROC and still get output that comes from these preliminary events. When a PROC is included, ODS calls an event that will generate a title if a title exists, a byline if it exists (if the PROC code contained a BY statement), and soon after, an event that generates

markup that corresponds to the beginning of a table. If the PROC generates five columns, ODS will call the column header event five times. If you ask for a header to span multiple columns, ODS will call for the appropriate events the appropriate number of times. After column headers, events are called to generate markup containing actual data. At this point ODS is dynamically calling events dictated by PROC results.

More specifically, what's dynamic is the number of times events are called, if at all, but what is not dynamic is the order in which events are called. This is something we count on from ODS - an order of event calls that corresponds to the order in which markup should appear. For example, we count on the SYSTEM_TITLE event which gives us access to the title in the TITLE statement to be called before events that generate PROC results. We expect events with byline information to be called between events with titles and events with table information, and we expect events that carry table header information to be called before those that carry table body information. Not only that, but we also count on ODS to help us generate properly nested markup.

Maybe one of the most compelling reasons to use ODS over other methods is the tagset's ability to write nested markup. As an example, consider the root element which, by definition, contains nested within it all other elements, usually at different levels. We know that while one event will generate "<ODM", the rest of the ODS output is generated through multiple event calls, many of which are called dynamically according to factors such as PROC results. The nature of nested markup is that markup in one part of the document is naturally tied to markup in a later part, so how can we build this relationship between two separate areas of the markup into the event model that generates it? How can we tie the generation of "</ODM>" to the generation of "<ODM" (or in general, any end tag to its corresponding start tag)? The answer is through event *states*.

When a tagset defines an event in states, then event calling begins to take on a nested pattern. Specifically, when a tagset defines a *start state* and a *finish state*, then a call to this event will execute only the code in the start state at first. Events that ODS deems to be *nested within* this event, or that carry markup to be nested within the tags generated with the start state of the "parent event", will then be called, and after their code finishes executing, the finish state of the parent event will then be executed. Conveniently, like the pre-determined order of event calling, the nesting pattern that ODS uses to call events is an intelligent one, meaning that the pattern closely resembles the pattern of nesting found in markup files. For example, the first event that ODS calls (starting in SAS 9.1.3) is called Initialize, and the second is called Doc. The Initialize event, like an empty element, has no event calls nested within it. If defined with start and finish states, the code in the finish state will execute immediately after the code in the start state, and so there is no reason to define this event in your tagset with states. Correspondingly, as we know, XML files (and sometimes HTML files) have declarations at the beginning of the document. Since these also have no nesting, one might use the Initialize event to generate the XML declarations. In terms of nesting, the Doc event is to event calls as the root element is to the markup. Just as the root element opens at the beginning (after the XML declarations) and closes at the end of the document, all event calls are nested within the call to the Doc event. This means that while code such as `put "<ODM"` found in the start state will generate markup in the beginning of the file, code such as `put "</ODM>"` found in the finish state will generate markup at the end of the file.

Example 12: Initialize and Doc event definitions

```
define event initialize ;
  put "<? xml version="1.0" encoding="windows-1252" ?>" ;
  put "<?xml-stylesheet type="text/xsl" href="define1-0-0.xsl"?>" ;
end ;

define event doc ;
  start:
  put "<ODM" ;

  finish:
  put "</ODM>" ;
end ;
```

Other examples of this event nesting exist too. The Doc event has two child events - a shorter Doc_head and a longer Doc_body - similar to the GlobalVariables and MetaDataVersion children of the Study element in Define. Inside Doc_body is the call of the Proc event corresponding to the beginning of PROC processing. Within the Proc event is a Table event, and within that are Header event calls for each column. After the Header events end, a Row event is called for each row of the table, and within each is a Data event for each cell.

Up to this point we've seen that ODS creates files by calling events that define markup generation in a pattern that resembles common markup structure, and that the tagset is a template for defining such events. Traditionally such a template has been a part of SAS's source code, but is now available to SAS programmers through PROC TEMPLATE. The problem is that SAS programmers don't know the names of the events that ODS calls, the order in which they are called, or their nesting pattern. Adding to the problem is the fact that the events called sometimes depend on PROC results and global settings. We've mentioned a few here, but for a programmer to be able to

associate a piece of markup with a particular event, we need complete documentation. Oddly enough, we can get this kind of information from any one of a class of tagsets called *mapping tagsets*.

Consider the mapping tagset illustrated in Example 13 below.

Example 13: a mapping tagset to document events and their call order

```
proc template ;
define tagset eventorder ;
default_event = "all" ;
indent=3;

define event all ;
  start:
  put event_name " (START)" nl ;
  ndent ;

  finish:
  xdent ;
  put event_name " (FINISH)" nl ;
end;
end;
```

Mapping tagsets like the one in example 13 are not used for creating deliverable output like Define.xml, but instead they help us learn about system and SAS metadata (not clinical metadata) such as event names. In this case we're defining a tagset called Eventorder that defines only one event called "all". As we know, ODS will call for several events, none of which are defined in this tagset, but because "all" is declared to be a default event (with the DEFAULT_EVENT= statement), its code will be executed as a substitute for undefined events. Note that each PUT statement combines literal text with a variable reference. We'll discuss *event variables* in more detail soon, but for now, know that its value is the name of the event being called. As in the DATA step, literal text is contained in quotation marks. Though it looks like another variable reference, "nl" moves the pointer to the next line. Following the PUT statement in the start state is an NDENT statement which moves the pointer to the right three spaces (as dictated by the INDENT= statement, used for readability). Similarly, the XDENT statement in the finish state moves the pointer to the left three spaces. We now put this tagset to practice in the following manner.

```
ods markup tagset=eventorder file = "proc print event order.txt" ;
title "tagset testing" ;
proc print noobs data = sashelp.class(obs=2) ;
var name sex age height weight ;
run;
ods markup close ;
```

The resulting file can be found in appendix A. What may first stand out about this output is the fact that it looks nothing like what you expect from PROC PRINT. Maybe what's most noteworthy though comes at the end. We can see from the PROC PRINT code that this will produce a table with two data rows (plus a header row) and five columns. At the end of the output file we see an event called Table_head being called. Nested within that is a call to the Row event, and nested within that are five calls to the Header event - one for each column header. After that, the Row event finishes, and because the header will have only one row, the Table_head event finishes. Then comes the Table_body event. Note that within this event call are two calls back to the Row event, each with five calls to the Data event nested within them. The two Row calls correspond to the two data rows of the table, and again, the five Data calls to the five columns of data. Once Table_body finishes, all other events in which it was nested finish too.

EVENT VARIABLES

We mentioned earlier that markup data is available through variables and since then we've seen examples of them. Style definitions from style templates make their way into markup by way of style variables. EVENT_NAME, whose values, as illustrated in example 13 above, reflect the name of the current event being called, is an event variable that might fall under the metadata umbrella. Other metadata variables include SASVERSION, ENCODING, DATE, and TIME. Many of these may not be useful in generating markup, but they are available and remain static across event calls. Other variables such as VALUE and NAME do carry dynamic data to be inserted into the markup and have the potential to change with every new event call. In all, between style and event variables, ODS has access to more than 500 variables. Though many are documented in the Online Documentation, many questions are still outstanding. How do I capture something global like titles? Do they populate a variable in the first event call and then remain static across all event calls like some metadata variables, or do they become available at just one event call and then disappear? If the latter, which event call? What other metadata is available? How do I know which cell

of a table a data value represents? Earlier we saw how a mapping tagset can serve as documentation about events and their call order. We'll now create another to document event variables and values.

Consider the mapping tagset illustrated in Example 14 below.

Example 14: a mapping tagset to document event variable population

```
proc template ;
define tagset allvars ;
default_event="all" ;

define event all;
put "EVENT:  " event_name NL ;
put "=====" NL ;
putvars event _name_ " = " _value_ nl ;
put "=====" NL ;
put NL ;
end;
end;
```

Once again we have a tagset with just one event defined, but because it's deemed to be a default, its code will be executed at every event call. Here we use the PUTVARS statement with the EVENT keyword. For any given event call, SAS will cycle through all of the event variables with non-missing values for that call and write to the file what follows the EVENT keyword. Note that once again, this combines literal text - the equal sign and surrounding spaces enclosed in quotes - with variable references `_NAME_` and `_VALUE_`. At any given iteration, `_NAME_` is the name of the variable at that iteration and `_VALUE_` is its value. An excerpt of the output generated using this tagset and the PROC PRINT code above is illustrated in Appendix B.

Though Appendix B is not the complete output file, we can start to gain an appreciation for the role that the variable VALUE plays. To answer our question regarding the availability of the title, we see that in the SYSTEM_TITLE_SETUP event call, VALUE holds the title. In the Proc_Branch event, VALUE holds the name of the PROC used. With each call to the Header event nested within Row nested within Table_head, VALUE contains the column headers, and in the Data event calls, it contains results of the PROC. For the most part, though more than 500 variables exist, VALUE contains most of the text that would be seen through a browser.

Keep in mind that this example comes with certain initial conditions - PROC PRINT with two observations and five variables, the NOOBS option on the DATA statement, one title, etc. The reader is encouraged to modify these conditions and observe the effect on the output. For example, try labeling the variables, use a BY statement, sum numeric variables, use an ID statement, use different system options, try with different PROCs or multiple PROCs. Also try modifying the mapping tagset. If the PUTVARS statement produces too much unnecessary output, try modifying the first mapping tagset by adding the value of the VALUE variable as illustrated below in example 15.

Example 15: adding VALUE to the EVENTORDER tagset

```
define event all ;
  start:
  put event_name " (START) VALUE=" value nl ;
  ndent ;

  finish:
  xdent ;
  put event_name " (FINISH)" nl ;
end;
```

In addition to the variables provided to us by SAS, tagset authors have the ability to create their own variables called *memory variables*. Unlike data set variables that are created with statements that have no keywords, memory variables are created with the EVAL statement (numeric variables) and the SET statement (character variables). Memory variables are always referenced with dollar signs.

Example 16: creating memory variables

```
set $charvar "character variable" ;
eval $numvar index($charvar, "r") ;
```

List variables and Dictionary variables are two special types of memory variables that work like arrays. Similar to the DATA step array, list variables associate a numeric index with a string while dictionary variables associate a string

with another string. For list variables, this is accomplished with the SET statement and a set of square brackets that either encloses a number to which the string is being associated, or nothing, which simply assigns the string to the next available numeric index in the list. For dictionary variables, the index is a quoted string. With both types of arrays the index can be replaced with a variable reference that resolves to an integer or a quoted string. Unlike DATA step arrays, no declaration of the dimension of the array is needed.

Example 17: list and dictionary variables

```
set $listvar[1] "first list index" ;
set $listvar[2] "second list index" ;
set $listvar[] "third list index" ; /* Added to the end of the array */
set $listvar[$numvar] "numvarth index" ;
set $dictionary ["one"] "first dictionary index" ;
```

GETTING STARTED ON YOUR DEFINE TAGSET

We've now been through the theory and are ready to put it all to practical use. We'll do that by taking another look at Example 12, but with a few changes.

Example 18: Initialize and Doc events revisited

```
define event initialize ;
    put "<? xml version="1.0" encoding=" encoding ">" ;
    put "<?xml-stylesheet type="text/xsl" href="define1-0-0.xsl"?>" ;
end ;
define event doc ;
    start:
        put "<ODM>" ;
        put "<Study>" ;

        finish:
            put "</Study>" ;
            put "</ODM>" ;
end ;
define event doc_head ;
    start:
        put "<GlobalVariables>" ;

        finish:
            put "</GlobalVariables>" ;
end ;
define event doc_body ;
    start:
        put "<MetaDataVersion>" ;

        finish:
            put "</MetaDataVersion>" ;
end ;
```

Notice that in this example, we are not writing out any of the attribute specifications for the root ODM element, the Study element, or the subelements of the GlobalVariables element. These observations speak to the primary focus of discussion, but before we get into those details, let's make a few other observations.

We first note that in the first declaration, the value of the encoding attribute is now an event variable reference rather than quoted literal text. This suggests that the event variable has a meaningful value at the time that the Initialize event is called. This variable in particular is a system metadata variable whose value remains consistent across all event calls. Values of other event variables such as VALUE and NAME aren't so predictable.

We've also added a few more PUT statements to more events that write out opening and closing Define.xml tags. Before we move on, it's important that we closely examine the event structure of example 18 and remind ourselves how this produces the element structure required by Define. In order to do that, we re-visit PROC PRINT output in Appendix A generated from the tagset illustrated in Example 13.

In example 18 we added PUT statements to add three more tag pairs, but why did we put them where we did? More specifically, why did we put one of them in the Doc event that also writes out the opening and closing of the root element, and the other two in separate events? Why did we choose the events we did for these tag pairs? We can

easily answer these questions by comparing the event call pattern (order of event calls and their nesting structure) in Appendix A to the element structure (order of elements and their nesting structure) required by Define.xml. Define starts with XML declarations that have no nesting. Likewise, when an ODS destination is opened, it begins by calling the Initialize event that has no nested event calls within it, making Initialize a good candidate for generating these declarations. After Define's declarations is the opening of the root element, which of course doesn't close until the end of the document. The Study element opens after the root element opening tag and then closes immediately before the close of the root element. Likewise, after executing statements in the Initialize event, ODS then calls the Doc event, inside of which all other event calls are nested, making Doc a good event to generate the opening tags of the ODM and Study elements in the Start state and the end tags in the Finish state. GlobalVariables is a child element of Study, but unlike Study's relationship to ODM, GlobalVariables is not an only child - MetaDataVersion is GlobalVariable's one and only sibling. Because, according to Appendix A, the Doc event call is the only "child event call" of ODM, generating these tag pairs with the Doc event would not have supported this sibling relationship. On the other hand, the Doc event call does have two nested sibling event calls - Doc_Head and Doc_Body - that correspond to the two child elements of Study. For that reason, we used these events in example 18 to generate these tag pairs.

Before we rely too heavily on the structure suggested by Appendix A, it's important that we keep in mind the conditions under which this was generated. An important question is how different Appendix A would look under different conditions. Recall that this was generated with a single PROC PRINT, the NOOBS option, reading only two observations and reporting on specific variables in a specific order indicated in the VAR statement. A title was also provided. For our purposes, this set of conditions is close to what we want to achieve - open the destination, PROC PRINT *each* data set of the MDDB (in the correct order), and close the destination. The most significant departure from Appendix A conditions is the use of multiple PROC PRINTs. We'll briefly discuss here how event calls would change from a logical point of view, but the reader is encouraged to experiment and see for themselves.

We mentioned earlier that initial event calls are relatively static, but then get more dynamic when the PROCs get processed. We also mentioned that because these static events are called before PROCs are processed, we can actually open an ODS destination and close it without any PROCs. In other words we can think of these events as being associated with the opening of the ODS destination. We can see from Appendix A that these initial events include precisely the events we defined in example 18 above. Once ODS is finished opening the destination, then it stands to reason that it's ready to start processing PROCs. Correspondingly, as seen in Appendix A, the first and only child of the call to the Doc_Body event is a call to the Proc event. It's at this point that event calling gets more dynamic. Nested within Proc are events called, for example, when a title is used and when a BY statement is used. The number of nested Row event calls depends on the number of rows that result (in this case, specified by the OBS= option in the PROC statement), plus another for the header. Nested within each Row event call are calls to the Data event, the number of which corresponds to the number of columns in the table. In this case, that's a function of the number of variables named in the VAR statement, plus the use of the NOOBS option in the PROC statement. Once enough Row events are called to complete the construction of the table, ODS finishes the processing of the PROC by executing the finish states of events nested within the Proc event. Once this is over, two things can happen. One is the closing of the destination, which corresponds to the call of the finish states of the initial static events. We see this in Appendix A. The other option is that another PROC can be processed, which would correspond to a new call to a new Proc event, and the PROC processing events start over again with the second PROC.

So far, things have worked out pretty well for us. Up to this point, the pattern that ODS uses in calling events, and in particular, the pattern of nesting, corresponds exactly to the pattern of elements and their nesting in Define.xml. We can be confident that the opening tags in the beginning of Define as well as their corresponding end tags will be generated only once - at the beginning and the end of the file - because the instructions for writing the text are in events that correspond to the opening and closing of the ODS destination, each of which we know happens only once. It's tempting to want to continue this pattern of event definition and see how much of the rest of Define (e.g. the ItemGroupDefs) we can accurately generate in this manner, but we soon discover that because of the variety in structure across child elements of MetaDataVersion, some adjustments will be necessary.

For starters, with a quick glance at Appendix A it appears that the pattern breaks down within DOC_BODY, for the simple reason that MetaDataVersion contains several child elements and DOC_BODY contains only one child event call to Proc. However we also just reasoned that Doc_Body will actually contain one child Proc event call *per PROC PRINT*, and so if the number of PROC PRINTS matches the number of child elements of MetaDataVersion, then maybe we have restored hope of having an event call structure that exactly matches Define's element structure. This also suggests one possible structure for your MDDB - each data set carries metadata for exactly one child element instance of MetaDataVersion. With this MDDB structure, we simply define the Proc event in the tagset to write out the markup for each child element of MetaDataVersion, and then by running PROC PRINT on each data set in the MDDB we get the right number of child elements.

One obvious problem with this solution as it is stated here is that we don't want to write out the same markup for each child of MetaDataVersion because different groups of these child elements have different element and attribute structures. For example, the def:AnnotatedCRF element contains no attributes and one or more def:DocumentRef

child elements, while the def:leaf element contains two attributes and exactly one def:title child element. ItemGroupDef contains several attributes and several instances of the ItemRef child element, and each CodeList element contains attributes as well as multiple levels of nesting. This means that if we're going to try and generate these elements from the Proc event, then that event definition will have to contain conditional PUT statements based on which child element (or group of elements) it is generating.

Conditional logic in tagsets is no problem (though the syntax is different from the DATA step's conditional logic), but we run into another bump in the road. Suppose we have a data set in our MDDB that contains all of the metadata that belongs in the ItemGroupDef element associated with the AE data set. This includes attributes of the ItemGroupDef element as well as attributes of each ItemRef child element. In the Proc event we can easily generate the literal text "<ItemGroupDef" and we can even generate the name of the first attribute plus the equal sign, but if the value of this attribute is contained in a data set of the MDDB that we are PROC PRINTing, then according to Appendix B, we don't yet have access to this value. In fact, as we pointed out earlier, we don't have access to it until the Data event is called, which is several levels of event call nesting deep within the call of Proc. This means that if we choose to generate markup from the Proc event, we are limited to generating literal text up until the occurrence of the first PROC result, and that to generate the PROC results, we'll have to do so with the Data event.

We started this discussion and developed this approach as a way to generate the child elements of MetaDataVersion, but it could also be one way to generate some of the markup that we skipped in example 18 - in particular, the ODM and Study attributes and the values of the child elements of GlobalVariables. Recall in that example that we had an event call structure that perfectly matched the element structure in the header of Define. Suppose now that we decide to store the ODM attributes in a data set of the MDDB for PROC PRINTing. Because we can't get PROC results until the Data events start getting called, we can't generate any of the header markup that follows these attributes until the Data events. That means we would have to remove the "<Study>" text as well as the entire Doc_head and Doc_body events. At most, you could use the start and finish states of the DOC event to generate the opening and closing of the root element (ODM). Optionally, you might also choose to use the finish state of Doc to generate the closing tags of the Study and MetaDataVersion events to make sure they close at the end of the file.

This is certainly one viable option, but because you're removing that assurance you had that the opening and closing tags were going in the right places, you'll need to take caution in how and when you generate these with this approach. The positive side is the use of the SAS data set for keeping such metadata. On the other hand, as a developer you might look at the ODM attributes and decide that across studies, they stay static enough to hard-code them into your tagset and not store them in a data set. This is another viable option that runs the same risk that hard-coding runs in any application - a decline in flexibility and an increased likelihood of having to change application code. The positive side is that you can continue to use the event definitions of example 18.

Finally, one other option that offers more flexibility than hard-coding, allows the event structure of example 18, and involves no storage in a data set is macro variables. Like other templates, the tagset language has the MVAR statement which allows you to declare macro variables that are to be resolved when the tagset is used rather than when it is compiled. Of course this would require the initialization of such macro variables before opening the ODS destination (e.g. %LET). References to such macro variables in the tagset definition do not include ampersands.

We noted earlier that different children of MetaDataVersion have different element and attribute structures. Because of that, generating these children requires conditional logic. We now conclude this section by looking at some examples that demonstrate this. In example 19, suppose we have a data set in our MDDB called "ACRF" that will be used to generate our def:AnnotatedCRF element. Because this element contains only one piece of data - the leafID attribute, which is actually an attribute of the def:DocumentRef subelement - our data set contains only one variable to hold all such attribute values. Running PROC PRINT on this data set, we know that the event variable VALUE will contain the value of that one data set variable when the Data event is called. Example 19 below demonstrates how we might define the Table and Data events to generate this part of the markup.

Example 19: generating the def:AnnotatedCRF element

```
define event proc ;
start:
put "<def:AnnotatedCRF>" nl / if $tablename eq "ACRF" ;;

finish:
put "</def:AnnotatedCRF>" nl / if $tablename eq "ACRF" ;;
end;

define event data ;
putq "<def:DocumentRef leafID=" VALUE "/>" nl / if $tablename eq "ACRF" ;
end;
```

Several parts of this example are worthy of note here. Note the syntax of conditional statements. Note also the new PUTQ statement which wraps double quotes around the resolution of the variable references (in this case, around the value of VALUE in the Data event definition). Finally, note the condition based on \$tablename. Remember that dollar signs precede memory variables, or variables created by the developer in the tagset definition. In this case, the memory variable was created in an earlier event that captured the name of the data set being processed.

Example 19 illustrates the generation of a relatively simple element. The simplicity comes from the fact that though AnnotatedCRF has a child element, only one piece of metadata is present, and so the data set in the MDDB only needed one variable. This meant only one call to the Data event for each Row call, and so we knew that when Data was called, the value of the event variable VALUE always represented the one leafID attribute. In example 20 below we illustrate the more complex def:leaf element which contains three pieces of metadata - two attribute values for def:leaf plus an element value for the def:title child element. We'll assume that a data set in the MDDB called "LEAFID" has three variables that hold this metadata. Furthermore, for reasons we'll see soon, we'll name these three data set variables for the attributes and elements they represent, replacing colons (illegal in variable naming) with underscores. That gives us the variables ID, xlink_href, and def_title. Note also that the letter casing of the variables is important.

The difficulty of having multiple variables corresponding multiple calls of the Data event within each Row event call, is that as the developer we have no control over the order in which these variables become available, which reflects their order in the data set. In this case, because def:title is a child of def:leaf, we need to be able to write out the def:leaf element and its attributes before we write out the def:title value. If this def:title value is not the third of the three variables in the data set, then rather than writing it to the output file right away, we need to hold on to it until we're ready to write it. This brings us to another approach to writing PROC PRINT results to the file - rather than writing them out as we receive them in the Data event, use the Data event to store them in a dictionary variable. Since we know that we've processed an entire row when we hit Row's finish state, we can write them out from here.

Furthermore, we can use the fact that we named the data set variables for the attribute and element names to create a dictionary variable whose string indices are easy to remember. After all, if we assign the PROC results to simple memory variables like \$a1, \$a2, and \$a3, we still wouldn't know which of these holds, for example, the ID attribute. However, with PROC PRINT, the value of the NAME event variable in the Data event is the name of the data set variable being processed. We therefore use each call of the Data event to associate a data set variable's name with its value. At Row's finish state when these are all collected, we write them out the way we need to.

Example 20: Creating the leafID element using a dictionary variable

```
define event data ;
set $data_values[name] VALUE ;
end ;

define event row ;
start:
unset $data_values ;

finish:
break / if section ne "body" ;
do / if $tablename eq "LEAFID" ;
put "<def:leaf ID=" $data_values["ID"] "xlink:href=" $data_values["xlink_href"] ">" ;
put "<def:title>" $data_values["def_title"] "</def:title>" ;
put "</def:leaf>" ;
done ;
end;
```

Let's analyze example 20 first by noting some new statements. UNSET followed by a variable name simply clears that variable of any values. BREAK means to exit the event. Statement blocks begin with DO and with DONE instead of the DATA step's END. We also notice a reference to the SECTION event variable. The Row event is called not only for data rows but also header rows. The code following the conditional BREAK in Row's finish state should only be executed for data rows, not header rows, which is why this conditional BREAK is there. We also note that in the Data event, the index is not an integer (list variables) or a quoted string, but rather a reference to an event variable. The index becomes whatever this variable resolves to during that call of Data. For example, if the first variable in the data set LEAFID is def_title, then def_title is the value of NAME during the first call to the Data event. If in the data set, the value of def_title is "Annotated CRFs #1", then as we know, this becomes the value of VALUE during the same Data call, and so \$data_values["def_title"] = "Annotated CRF". The beauty of this approach is that even if def_title is the first variable in the data set, the fact that it is written last doesn't matter because before anything is written, all three data values are stored in the dictionary variable, each associated with a string that matches the name of the data set variable that held its value. At the end of the row everything is written out and the slate (dictionary variable) is wiped clean and the process starts again for the next row.

Finally, unlike example 19, the element name in this example is generated along with its attributes and child elements, rather than in the PROC event. That's because the MDDB structure was such that each observation in LEAFID represented a new def:leaf element, whereas the observations in ACRF each represented a def:DocumentRef child element.

The techniques described in example 20 as well as any others used throughout this paper are among many that you as a developer can use to generate your markup. The purpose of this section of the paper is to demonstrate how the event model works and how to use event definitions in your tagset not only to write markup to the file but also to capture event variable values when they're available. The techniques illustrated above are not claimed to be better than any others, but rather were used as a means to fulfill this purpose. In the meantime, they also illustrated tagset tools that are available to you, as well as some of the difficulties involved in directing ODS to deliver output.

In fact even the use of ODS isn't necessarily the unanimous choice for SAS tools. Other developers have made creative use of the macro facility and the DATA step to accomplish the same purpose. For me there were three reasons for choosing ODS. One reason is that SAS is working closely with CDISC to provide support for Define.xml, and that support will come in the form of ODS tagsets. Different companies will make different decisions when that time comes; those that convert to using SAS's new tools from using their own tagset will have spent valuable time considering things like MDDB structure, the inner workings of ODS, and the ability to modify SAS's new tagset if necessary; those that continue to use their own can still learn from what SAS has produced. A second and more technical reason is ODS's ability to incorporate complex levels of nesting with the event model and in particular, start and finish states. And finally, in some ways to me it feels more natural. With the level of responsibility for our metadata that CDISC requires, organized storage in a database is necessary. Define.xml can be thought of as a copy of your MDDB in a different file format. SAS programmers sometimes make copies of data sets with PROC PRINT. With time spent up front for development, users can do what's natural to them - manipulate data and PROC PRINT it.

Although it isn't *completely* natural. We have provided no functionality for titles and footnotes or PROC PRINT features like BY and ID statements. This tagset was written in a box for one purpose only. Also, we noticed at times the dependency of the tagset code on external factors. We assumed data set variable names that matched attribute and element names. We also assumed that the order of the PROC PRINTs will correspond to the order of Define's elements. Such restrictions on the use of an ODS destination is something we're not used to, but then again, most ODS destinations create markup for tabular output - markup that is relatively uniform, unlike the varied nature of MetaDataVersion's children. This tagset can only generate valid Define.xml if the environment is set up right. This suggests that the tagset development is tied in with the structure of the MDDB and each of the data sets it contains. For that reason, some may feel that to ensure the right environmental conditions, use of the tagset must be part of a bigger application, perhaps wrapped in a macro. These are decisions to be made by the development team and the expected users.

CONCLUSION

The challenges a SAS programmer faces in the pharmaceutical industry can be daunting, depending on one's experience. They present opportunities for learning technologies that can't be expected to be learned over night. This paper makes no claim that you will be generating perfectly valid and compliant Define.xmls immediately after reading it. Instead, in addition to other resources including the references mentioned at the end, it is intended to be one of several resources, and hopefully one that is condensed enough to be one of the first ones you reach for. The specific techniques and assumptions used may or may not be suitable for your environment, but hopefully they at least provided you the tools to develop your own.

REFERENCES

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

Molter, Mike "A Tiptoe Through the Tagset Field." *Proceedings of the SAS Global Forum Users Group International Conference*, April 2008

Clinical Data Interchange Standards Consortium. "Case Report Tabulation Data Definition Specification (define.xml)", February 2005 http://www.cdisc.org/models/def/v1.0/CRT_DDSpecification1_0_0.pdf

Harold, Elliotte Rusty and Means, W. Scott. (2004) "XML in a Nutshell". Sebastopol, CA: O'Reilly Media, Inc.

CONTACT INFORMATION

Please feel free to contact me with questions and comments.

Mike Molter
INC Research

(919) 926-5710 (work)
(919) 414-7736 (mobile)
mike.molter@nesug.org

SAS and all other SAS Institute Inc. product or service names are registered trademarks or trademarks of SAS Institute Inc. in the USA and other countries. © indicates USA registration.

Other brand and product names are trademarks of their respective companies.

APPENDIX A

The following PROC PRINT output using the tagset from example 13 illustrates the order of event calling and the event nesting pattern.

```
initialize (START)
initialize (FINISH)
doc (START)
  doc_head (START)
    doc_meta (START)
    doc_meta (FINISH)
    auth_oper (START)
    auth_oper (FINISH)
    doc_title (START)
    doc_title (FINISH)
    stylesheet_link (START)
    stylesheet_link (FINISH)
    javascript (START)
      startup_function (START)
      startup_function (FINISH)
      shutdown_function (START)
      shutdown_function (FINISH)
    javascript (FINISH)
  doc_head (FINISH)
  doc_body (START)
    proc (START)
      anchor (START)
      anchor (FINISH)
      page_setup (START)
        system_title_setup_group (START)
          title_setup_container (START)
            title_setup_container_specs (START)
              title_setup_container_spec (START)
              title_setup_container_spec (FINISH)
            title_setup_container_specs (FINISH)
            title_setup_container_row (START)
              system_title_setup (START)
              system_title_setup (FINISH)
            title_setup_container_row (FINISH)
          title_setup_container (FINISH)
        system_title_setup_group (FINISH)
      page_setup (FINISH)
    system_title_group (START)
      title_container (START)
        title_container_specs (START)
          title_container_spec (START)
          title_container_spec (FINISH)
        title_container_specs (FINISH)
        title_container_row (START)
          system_title (START)
          system_title (FINISH)
        title_container_row (FINISH)
      title_container (FINISH)
    system_title_group (FINISH)
  proc_branch (START)
    leaf (START)
    page_anchor (START)
```

```

page_anchor (FINISH)
output (START)
table (START)
  rowspec (START)
    cellspec (START)
    cellspec (FINISH)
    cellspecsep (START)
    cellspecsep (FINISH)
    cellspec (START)
    cellspec (FINISH)
    cellspecsep (START)
    cellspecsep (FINISH)
    cellspec (START)
    cellspec (FINISH)
    cellspecsep (START)
    cellspecsep (FINISH)
    cellspec (START)
    cellspec (FINISH)
    cellspecsep (START)
    cellspecsep (FINISH)
    cellspec (START)
    cellspec (FINISH)
  rowspec (FINISH)
  colspecs (START)
    colgroup (START)
      colspec_entry (START)
      colspec_entry (FINISH)
      colspecsep (START)
      colspecsep (FINISH)
      colspec_entry (START)
      colspec_entry (FINISH)
      colspecsep (START)
      colspecsep (FINISH)
      colspec_entry (START)
      colspec_entry (FINISH)
      colspecsep (START)
      colspecsep (FINISH)
      colspec_entry (START)
      colspec_entry (FINISH)
      colspecsep (START)
      colspecsep (FINISH)
      colspec_entry (START)
      colspec_entry (FINISH)
    colgroup (FINISH)
  colspecs (FINISH)
table_headers (START)
  header_spec (START)
    sub_header_colspec (START)
      col_header_label (START)
      col_header_label (FINISH)
    sub_header_colspec (FINISH)
    sub_header_colspec (START)
      col_header_label (START)
      col_header_label (FINISH)
    sub_header_colspec (FINISH)
    sub_header_colspec (START)
      col_header_label (START)
      col_header_label (FINISH)
    sub_header_colspec (FINISH)
    sub_header_colspec (START)
      col_header_label (START)
      col_header_label (FINISH)
    sub_header_colspec (FINISH)
    sub_header_colspec (START)
      col_header_label (START)
      col_header_label (FINISH)

```

```

    sub_header_colspec (FINISH)
    header_spec (FINISH)
table_headers (FINISH)
table_head (START)
  row (START)
    header (START)
    header (FINISH)
    header (START)
    header (FINISH)
    header (START)
    header (FINISH)
    header (START)
    header (FINISH)
    header (START)
    header (FINISH)
  row (FINISH)
table_head (FINISH)
table_body (START)
  row (START)
    data (START)
    data (FINISH)
    data (START)
    data (FINISH)
    data (START)
    data (FINISH)
    data (START)
    data (FINISH)
    data (START)
    data (FINISH)
  row (FINISH)
  row (START)
    data (START)
    data (FINISH)
    data (START)
    data (FINISH)
    data (START)
    data (FINISH)
    data (START)
    data (FINISH)
    data (START)
    data (FINISH)
  row (FINISH)
  table_body (FINISH)
  table (FINISH)
  output (FINISH)
  leaf (FINISH)
  proc_branch (FINISH)
  proc (FINISH)
  doc_body (FINISH)
doc (FINISH)

```

APPENDIX B

The following PROC PRINT output using the tagset from example 14 illustrates event variables and their values by event call.

```
EVENT: system_title_setup
```

```

=====
anchor = IDX
toclevel = 1
colcount = 1
event_name = system_title_setup
encoding = windows-1252
operator = mmolter
date = 2009-01-13
sasversion = 9.1
saslongversion = 9.01.01M3P02022006

```

time = 11:02:47
state = start
value = tagset testing
page_count = 1
total_page_count = 1
firstpage = 1
proc_name = Print
dest_file = body
bodyname = C:\Documents and Settings\mmolter\My Documents\SGF 2009\variable values.txt
tagset = putvars1
style = Default
javadate = 2009-01-13
javatime = 11:02:47-05:00
data_viewer = Report
style_element = SystemTitle

EVENT: proc_branch

=====

name = Print
label = The Print Procedure
anchor = IDX
toclevel = 1
colcount = 1
event_name = proc_branch
encoding = windows-1252
operator = mmolter
date = 2009-01-13
sasversion = 9.1
saslongversion = 9.01.01M3P02022006
time = 11:02:47
state = start
value = Print
proc_count = 1
total_proc_count = 1
page_count = 1
total_page_count = 1
proc_name = Print
dest_file = body
bodyname = C:\Documents and Settings\mmolter\My Documents\SGF 2009\variable values.txt
tagset = putvars1
style = Default
javadate = 2009-01-13
javatime = 11:02:47-05:00
data_viewer = Report
style_element = ContentProcName

=====

EVENT: data

=====

type = string
name = Sex
dname = Sex
label = Sex
anchor = IDX
colstart = 2
row = 2
colwidth = 1
scale = 0
precision = 0
colcount = 1
event_name = data
encoding = windows-1252
operator = mmolter
date = 2009-01-13
sasversion = 9.1
saslongversion = 9.01.01M3P02022006
time = 11:02:47
section = body
state = start
col_id = 2
value = M
output_name = Print
output_label = Data Set SASHELP.CLASS
proc_count = 1

```
total_proc_count = 1
page_count = 1
total_page_count = 1
proc_name = Print
data_row = 1
dest_file = body
bodyname = C:\Documents and Settings\mmolter\My Documents\SGF 2009\variable values.txt
tagset = putvars1
style = Default
sasformat = $F
unformattedtype = string
unformattedwidth = 1
javadate = 2009-01-13
javatime = 11:02:47-05:00
data_viewer = Report
style_element = Data
last_stacked_value = 0
first_stacked_value = 0
=====
```