Paper AD11

Using TFL Metadata to Populate Titles & Footnotes in SAS Programs for Clinical Trials

Stephen Hunt, ICON Clinical Research, Redwood City, CA
Brian Fairfield-Carter, ICON Clinical Research, Redwood City, CA

ABSTRACT

Compared with even the most routine data manipulations and calculations that go into the production of TFLs (Tables, Figures, and Listings), the SAS code necessary for inserting titles and footnotes seems fairly trivial. However, titles play a vital role in explaining what the body of a particular TFL is presenting, while footnotes add critical details regarding implementation of, and the occasional exception to, analysis specifics. Documents are essentially meaningless unless their content is qualified by explanatory titles and footnotes. Titles and footnotes can therefore be thought of as 'data about data', which is the definition of 'metadata'. Given the importance of this metadata, it's not surprising that it undergoes frequent revision during the course of a project. Because title and footnote revisions usually take place independently of modifications to calculations, programming algorithms, and formatting and are often imposed in concert across multiple TFLs, it makes sense to manage this metadata externally from SAS source code. This paper begins by advocating the use of external TFL metadata in a SAS programming context within the pharmaceutical/biotech industries, continues by describing a few common methods for metadata storage, and culminates this evolutionary progression at its logical conclusion with the use of Extensible Markup Language (XML) format. Additionally, a demonstration of a simple 'home-grown' application front-end for managing this XML metadata is provided along with a description of its integration with SAS's XML parser, as well as several macro techniques for querying and utilizing the metadata in TFL-producing SAS programs.

INTRODUCTION

This paper describes one particular technique of accomplishing an increasingly common practice whereby all elements within titles and footnotes used in all SAS programs for a particular deliverable can be entered and managed as an independent set of metadata during the course of program development. Various means of storage for this TFL metadata are explored, with an emphasis on the ubiquity of use and platform independence offered by XML. An HTML application (HTA) developed by the author's highlights some of the concepts and elements of TFL metadata storage and maintenance in a generalized fashion while providing a specific example of the potential for metadata management within a SAS programming environment.

EXTERNAL VERSUS INTERNAL METDATA

Titles and footnotes are ubiquitous in published works. They provide a necessary structure and context to everything from marketing advertisements to professional journal articles to spam emails. The final publication typically produced in a clinical trial is the study report, which, among other things, includes tables, figures, and listings replete with information organized in no small part by a structured series of title elements, each of which contains, at the very least, a unique number and brief description of the output set within a particular page or group of pages. For SAS programmers working to produce any individual component of the output destined for inclusion in the study report, information passed to SAS title or footnotes statements must necessarily either reside within the SAS program itself as quoted text, or external to a particular SAS program in a seemingly limitless format (e.g., binary data stored in an application such as MS Excel or Access, a simple comma-delimited text file, as a series of global macros defined within another included SAS program, etc…).

Although this paper ultimately promotes the external storage of TFL metadata, perhaps it's best to start with a brief discussion on the overwhelming shortcomings of internal title/footer storage. In general, keeping title and footer information within the parent SAS program almost always requires more maintenance and greater risk of errors than the alternative. For example, re-numbering of tables or listings within a series is virtually inevitable through the review process as reviewers cull out or add to the required set of final output. The consequence of storing a table number within an individual SAS program means that every program producing a subsequently numbered table must be updated with every table insertion or deletion. Besides the inconvenience of changing multiple programs where no other functional change was required, the very act of making *any* modification now changes the date/time stamp on all effected programs, thus requiring re-documentation of QC to keep auditors satisfied.

As another example, tables are occasionally modified based on their population subsets (e.g., a table formerly slated under the ITT Population may instead be requested to be presented based on an Efficacy Evaluable Set or vice versa). Such a modification requires at least 2 changes to the affected SAS program (i.e., Title, and subsetting where/if clause). The more elements within the SAS program that require modifications, the more likely an inadvertent programming error will occur. This is especially so given that most changes of the 'late breaking' variety are rarely allowed to proceed at a leisurely pace. Thus, it follows that the more linked elements that can be controlled externally (and thus modified *en mass* via a change to a single element) the less likelihood of errors.

Finally, many interim deliverables require program development by a blinded programmer, followed by production by a second, unblinded individual. Internally-stored title/footer changes in this context require re-copying files over into production areas (or re-checking out modified files from a version-control repository). External storage of these elements makes such a task completely unnecessary. Additionally, the ability to consolidate all the elements of a particular project-wide section of information into a single data source has obvious efficiency implications for the lead programmer managing the entire deliverable production process.

Due to the above mentioned shortcomings (and others beyond the scope of this paper), more and more programmers are beginning to make use of various 'external' means of storing TFL metadata.

ELEMENTS OF TFL METADATA

To make the successful break away from the practice of internally storing potential 'external' pieces of a typical clinical trial table requires an examination of what elements are crucial to uniquely identifying it, which ultimately reveals a link between this external data and the SAS program that it supports. A surprising amount of information can be effectively disassociated (and thus practically segregated) from the SAS program itself: Naturally, the titles and footers are the most obvious pieces, but elements such as the table or listing number, the population subset required, the particular production area (e.g., DSMB vs. Primary Study Report vs. Ad Hoc), and even the type of output produced (table vs. listing) are all elements that can easily be shifted away from the original SAS program itself. We'll explore the absolute minimum elements required to maintain this program/metadata association later in the paper. But first, let's explore the different mediums for TFL metadata storage.

EVOLUTION OF TFL METADATA STORAGE

Naturally, the first logical step away from a table-producing program containing TFL metadata elements is the storage of these elements within another SAS program. On the most basic level, such a program merely defines a set of uniquely-named, global macro variables that are instantiated via either a '%inc' call or a session invocation program run prior to program submission. A slightly more involved, but still SAS-based, means might also make use of a text file that gets read into SAS via a datastep and infile statement, then either converts the data directly into uniquely defined macro variables or into generic fields (e.g., &title1) that are re-defined at each record-level selection associated with each unique piece of output. Thus, the first easily identifiable 'progressive stage' of TFL metadata evolution involves individual records of the associated data being specifically linked with an individual piece of output (whether a unique or repeat table, listing, or figure). This invariably leads to the next level of organizational complexity – database storage.

Monopolistic considerations aside, most of us spend at least a portion of our time behind a desktop computer that uses Microsoft® Windows™. As such, probably the most commonly-adopted non-database 'database' in use is Microsoft Office™ Excel©. The rows/columns format and ease of data entry appeal to programmers and non-programmers alike. The SAS import facility (whether via CSV or EXCEL DBMS connections) allows for a relatively pain-free transference of Excel data into SAS datasets. However, the 'relatively' qualifier exists due to complications arising from Excel's lack of explicit data 'typing' (SAS has to figure out whether the values in each column are numeric or character, because Excel won't tell it) or occasional unexpected handling of special characters, just to mention a few. Of course, there's plenty of alternate ways (and numerous resultant papers) on importing from Excel to overcome virtually any bug discovered in one process or another. However, an arguably 'neater' option is the use of MS Access over Excel. Access has the benefit of being an actual database application, thus the explicit variable lengths and typing tend to make for more error-free imports into SAS datasets. Also, Access has the additional benefit of being able to more easily manage the data via Data Access Pages (DAP) – See Fairfield-Carter et. al., 2005. However, one of the unattractive features of maintaining metadata in both Access and Excel is their binary source format as a data storage mechanism (requiring larger storage space and making tracking and/or comparison of changes difficult – although MS Office 2007 is offering a wider variety of XML formatted output files). Another inconvenience is the problem of platform dependence (if you ultimately want to use this metadata on UNIX or LINUX, an extra step is required in converting the data into a SAS dataset on its way). Perhaps most irritating of all, the structures of both applications (i.e., horizontal rows of data) make for awkward modification to the data when working with more variables than will fit within one full screen-width (e.g., scrolling left and right through a series of up to 10 footnotes, each of which can be as long as the linesize of your SAS output, can be a bit of an eye-crossing experience).

Thus, with a view for overcoming as many of the above-mentioned limitations as possible, the remainder of this paper will focus on XML as the metadata storage medium of choice.

QUERYING AND USING METADATA IN A TLG PROGRAM

However, before we get into XML, let's take a step back first to briefly cover how it is that we can associate this external information with the SAS program(s) that will ultimately make use of it. As it turns out, only 2 elements are actually required to uniquely identify any table or listing metadata with its associated program. In our setup, both elements are made available as macro variables for processing within the macros alluded to below, but this is by no means the only method possible for making the appropriate associations.

The first is, quite simply, the SAS program name itself. In both UNIX and Windows, the program name can easily be identified automatically via the following code (alternatively, since program names rarely *need* to change, this value could be 'hard-coded' into the program itself - but what fun would that be?):

```
%let pgm_name=%sysfunc(getoption(SYSIN));

data _null_;
   length pgmname $200;

   *** THIS PARSES THE PROGRAM NAME FOR BOTH WINDOWS AND UNIX FROM THE FULL PATH NAME.;
   pgmname=scan(translate("&pgm_name","\","/"),-1,"\");
   pgmname=tranwrd(pgmname,".sas","");

   call symput('pgmname',trim(left(pgmname)));
run;
```

Once this is accomplished, the only need for a second key field for linking to the database is due to the possibility of multiple pieces of output being generated by the same SAS program. As such, an arbitrary field which we'll call SERIES will suffice. For our purposes, this field will contain the value '0' (null) for every record where a program does *not* produce multiple output files. We'll cover the possible values in the event of multiple output files later. In this instance, SERIES can be initialized within each call of an autoexec.sas included at the beginning of each program as…

```
%let series=0;
```

From here, the dynamics of the particular method used to insert the associated data from the specific record selected as linked to the program being run can vary depending upon need and creativity. The essential concept is calling a macro (or macros) at the beginning of the SAS program itself that passes in the 2 fields mentioned above, and thus identifies the specific record associated with the desired output. For example, the first few lines of an AE table might be thus…

```
%inc 'autoexec.sas'; *** MACRO VARIABLE SERIES SET TO '0'.;
%let pgmname=t_ae;
%m_tfl;
```

Where the macro %m_tfl.sas (which you've programmed in a directory path identified by your autocall library, of course) pulls the specific record from our metadata…

PGMNAME	TFLNUM	SERIES	TITLE1	TITLE2	POP
t_ae	38	0	Summary of Adverse Events		Safety Population

… and sets as many fields as needed/desired into macro variables for use in title and footnote statements. In this case, since the sample table only creates one piece of output, the simple record selection that follows accomplishes this link.

```
data null_;
   set tflspecs.tfls;

   if trim(left(upcase(pgmname)))=%upcase("&pgmname") & trim(left(upcase(series)))=%upcase("&series");

   *** DO MORE STUFF BELOW;
.....
```

Beyond this, it's really just a matter of setting up macro counters and loops for the specific number of titles and footnotes and for passing this information into SAS title/footnote statements. The following is an abbreviated example:

```
.... <CONTINUATION OF THE ABOVE DATASTEP>
   n_foot = 1;

   %do i = 2 %to 9;
      if foot&i ne "" then n_foot = &i;
      call symput("foot&i", trim(left(foot&i)));*** ASSIGN FOOTNOTES 2-9 TO MACRO VARIABLES FOOT2-FOOT9;
   %end;

   *** ADD 1 TO THE TOTAL TO MAKE ROOM FOR THE SOURCE/DATETIME FOOTER.;
   n_foot=n_foot+1;

   call symput('n_foot',compress(n_foot));
run;

%do i=1 %to &n_foot;
   %if &i=&n_foot %then %do;
      footnote&i j=l "¿Source: &user &ext_name     &systime. &sysdate9.";
   %end;
   %else %do;
      footnote&i j=l "&&foot&i";
   %end;
%end;
```

Naturally, the same can be applied to title lines. In the above example, the conspicuously absent &foot1 is actually pre-defined in autoexec.sas as a series of dashes in order to place a divider between the body of the output and its first footer. Beyond this, as much or as little as deemed appropriate by the programmer can be dynamically created within the macro call(s) using the values stored in the TFL metadata. For example, a dynamic population-subsetting where clause can be generated and applied in the following fashion.

```
%macro m_pop;

   %global m_pop;
   %let m_pop=;

   data _null_;
      length trtcode $100;
      set tflspecs.tfls(where=(trim(left(upcase(pgmname)))=%upcase("&pgmname") &
                               trim(left(upcase(series)))=%upcase("&series")));

      if trim(left(upcase(pop)))="SAFETY POPULATION" then do;
         trtcode="where saf=1";
      end;
      ..... <etc.>

      call symput('m_pop',trim(left(trtcode)));
   run;

%mend m_pop;

%inc 'autoexec.sas'; *** MACRO VARIABLE SERIES SET TO 'O'.;
%let pgmname=t_ae;
%m_pop; *** COULD BE CALLED WITHIN %M_TFL.SAS INSTEAD, IF DESIRED.;

proc sort data=derived.ae out=ae;
   by patid;
   *** SELECT BY THE PARTICULAR POPULATION NEEDED FOR THIS TABLE.;
   &m_pop;
run;
```

Naturally, this becomes most useful in a table that produces multiple files where the various output generated differs primarily (or entirely) by the population required for summarization.

On that note, it's a small step towards defining any type of replicate output within the same SAS program using the 2 primary key variables in our TFL metadata.

```
%inc 'autoexec.sas'; *** MACRO VARIABLE SERIES SET TO 'O'.;
%let pgmname=t_ae;

%macro rpt_table(series=);
   %m_pop;

   proc sort data=derived.ae out=ae;
      by patid;
      *** SELECT BY THE PARTICULAR POPULATION NEEDED FOR THIS TABLE.;
      &m_pop;
   run;

   <DATA SUMMARIZATION STEPS...>

   %m_tfl;
   <PROC REPORT/TABULATE SECTION>

%mend rpt_table;

%rpt_table(series=38.1);
%rpt_table(series=38.2);
```

Alternatively, one could use SERIES=1 when another metadata variable called TFLNUM (representing the table number itself) already equals 38 to avoid repetition. The point being, as long as SERIES is unique within the same program name, the particular architecture you use is up to you.

A USER-FRIENDLY GRAPHICAL INTERFACE FOR MANAGING TFL METADATA AS XML

XML, while terrific at storing information, is not really the most user-friendly to view and modify in its raw form. The typical 'generic' format of XML conforms to a nested, tag structure like the following.

```
<?xml version="1.0"?>
<TFLS>
   <PGMNAME>T_AE</PGMNAME>
   <TFLTYPE>TABLE</TFLTYPE>
   <POP>ANALYSIS POPULATION:SAFETY</POP>
   <TFLNUM>38</TFLNUM>
   <SERIES>0</SERIES>
   ...
</TFLS>
```

While a description of the history and variety of format mappings and XML parsers are beyond the scope of this paper, the 'take-home' points in favor of XML usage are its ubiquity and transparency. Most data applications worth their salt can read/write XML (including SAS, of course!). Additionally, the text basis for XML makes it ideal for use with even the simplest version control methods (i.e., archiving and text comparison with older versions).

However, in order to efficiently read and write XML (i.e., working with hundreds of lines of the above example is not a viable or sustainable option if one wishes to maintain sanity), it's best to make use of one of the many XML parsers available. As mentioned previously, the majority of SAS programmers still work to some extent within MS Windows. As luck would have it, the typical Windows install contains several XML parser options within reach of the intrepid programmer via some simple Windows Scripting Language techniques. Our particular application makes use of Microsoft's ADODB recordset object to read and write XML. To do so, we've couched some VBScript and JavaScript code within an HTML application (HTA) file (see Fairfield-Carter, Sherman, et. al., 2005).

While most people are probably familiar with HTML, 'dynamic' HTML (dHTML), and the subtle but important differences between dHTML and HTAs are probably less well-known. Dynamic HTML is simply HTML with embedded scripting elements. These scripting elements may perform a variety of functions, including operating on external files, dynamically generating HTML, and altering HTML styles. Because the possibility exists that these scripts may perform pernicious tasks, most web browsers include a large and complex set of security measures,

which often interfere with legitimate dHTML applications. HTA simply provides an alternative method of declaring dHTML, which circumvents these security measures. For the latest download of the application describe herein, please visit <http://sourceforge.net/projects/xmltop/>.

The code below, taken directly from the aforementioned HTA file, is a quick example of how to implement ADODB to read in an XML document that's in the format generally produced by MS Excel or Access (i.e. *not* the generic format shown above).

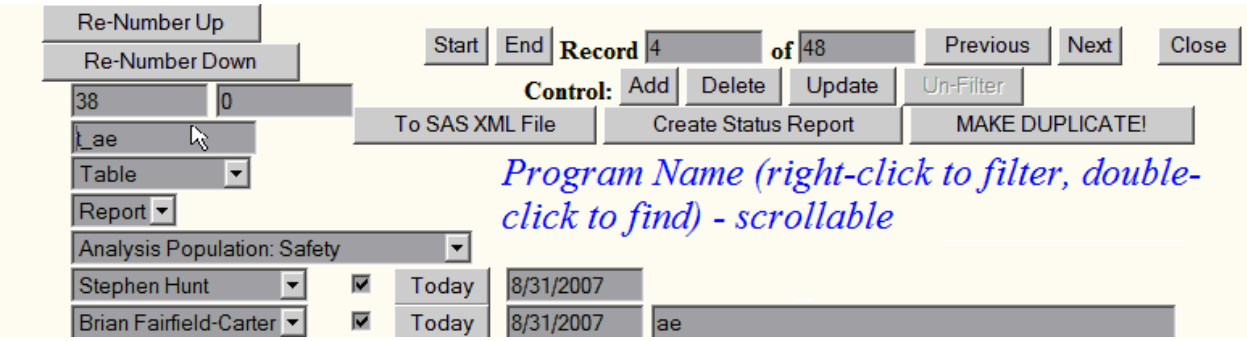
```
<script language=vbscript>

'Get the current directory location from the HTML application running.
xmlfolder=replace(fso.getparentfoldername(mid(window.location.pathname,1)),"%20"," ")
xmlfile = xmlfolder & "\TFLS.xml"

'create an new ADODB recordset object.
set rs = CreateObject("adodb.recordset")

'open the XML file into the recordset object (see MSDN ADO Recordset Open method for command param details).
rs.Open xmlfile,,2,3,256
...
</script>
```

The particular 'rowset' schema using ADODB has advantages and disadvantages over the generic format. The upside is that ADODB can read/write the XML file on a single record level, which means that updates to one record are independent events and have no impact on other records (which also means that other people can theoretically modify other records in the same document at the same time). The former is generally a good thing, because it forces the user to actively implement a change (recordset update) to a particular record, or, lacking that, no change will be made when the user moves to the next record. The alternative to this (using a different type of connection method not discussed here) is that any change made, whether intentional or accidental, changes the entire document by default unless the change itself is reversed. From a conservative perspective, it's arguably safer to require intentional modifications at each and every record rather than to have to worry if an errant keystroke was carried into the document inadvertently. Theory aside, the disadvantage to ADODB use is that any XML format other than the generic (as in the example above) requires an extra step for SAS to be able to translate with its own parser when using the XML libname engine. We'll address that more later on in the paper.

For now, the first, obvious advantage of an ADODB record-level access to any external database is that you can then tailor the viewing of this data to whatever the most user-friendly format would be for manipulation of the data itself. Recall the structural limitations of working with Excel (and Access when not using a Data Access Page) via the rows and columns stretched across the horizon. By opening and presenting the underlying data through any application front-end (we've chosen HTA, but plenty of others would do just as well), we can completely control the method of presentation. Specifically, we've chosen to view and modify our XML metadata in the same spatial orientation as it appears in the corresponding table shells (i.e., a vertical rather than horizontal structure). Here's a screenshot of a single record viewed vertically:

As you can see, all fields related to the particular metadata record are in full view, and are positionally analogous to their appearance on the actual SAS output.

The second item of note is that none of the fields are explicitly labeled. To maximize the efficiency of our workspace, we've added a JavaScript label function to each field that actives a text description in a specific area of the application upon the user scrolling over the field with his/her mouse:



Although the metadata elements contained in any file will vary dependent on whatever a particular programmer deems useful for their specific needs, our field selection (and recommendation) is as follows:

PGMNAME - The name of the SAS program producing the output
TFLTYPE - The type of output produced (e.g., Table, Figure, etc...)
AREA - The study area (e.g., primary, DSMB, adhoc, etc...)
TFLNUM - The Table or Listing number
SERIES - The unique identifier within PGMNAME – note that this is generally a sub-part of the table number itself

POP - The population (if applicable)
PGMER - The name of the original program author
PGMED - A binary flag for program completion
PGMDATE - The date of program completion
TESTER - The name of the QC program author
TESTED - A binary flag for QC completion
TESDATE - The date of QC completion
TITLE1 - The first title of the table/listing
TITLE2 - The second (optional) title
FOOT2 – FOOT9 – Up to 8 footnotes, assuming 1 'border' line and 1 source footer line.
SOURCE - Derivation datasets used in the SAS program

In addition to the data-driven metadata elements of this XML front-end are a number of control elements, generally in the form of buttons, although some features (such as filtering and find) are driven by right-clicking or double-clicking on particular fields. Naturally, of primary importance would be control buttons that allow for movement through the recordset itself. For example, the following is necessary for scrolling forwards within the database to the next record in sequence:

```
function movenext()

      'MOVENEXT IS AN ADODB.RECORDSET METHOD.
      rs.MoveNext

      'IF YOU SCROLL TO THE END OF THE RECORDSET, START BACK AT THE BEGINNING (OTHERWISE AN ERROR WILL OCCUR).
      If rs.eof=true Then
         rs.movefirst
       End If

      'POPULATE THE NEWLY SELECTED RECORD WITH THE APPROPRIATE VALUES.
      getvalues()

end function
```

At each new record, the function 'getvalues()' referenced above is required to reload all the values shown on the dHTML display with the contents of the new record that has been selected. In this case, the set of values in the recordset itself is stored in the same manner as an array, with the value of the first item being represented by the recordset element zero (0) and progressing incrementally upwards to the maximum number of fields minus 1. For instance:

```
function getvalues()

   on error resume next

   pgmname.value = rs(0)
   tflnum.value = rs(1)
   series.value = rs(2)
   ...
   source.value=rs(22)

end function
```

Once past the relatively basic navigational and data-manipulation functions (others include *delete, update, and save*, which writes changes made to the recordset back out to the originating XML document – see http://www.msdn.com for a more thorough explanation of the variety of functions available to ADODB.RECORDSET scripting), a slightly more advance example (again, in VBScript) for performing recordset filtering is as follows:

```
'FOR THE PARTICULAR ELEMENT, TURN OFF THE DEFAULT RIGHT-CLICK MENU.
window.event.returnvalue=false

'IF ANY FILTERING ALREADY IS IN PLACE, APPEND THE NEW ADDITIONAL FILTER ONTO IT.
if filterval<>"" then
   filterval=filterval & " AND " & window.event.srcElement.name & "=" & chr(39) &
             window.event.srcelement.value & chr(39)

   'OTHERWISE, CREATE A NEW FILTERING COMMAND STRING. CHR(39) REPRESENTS A SINGLE QUOTE.
   else filterval=window.event.srcElement.name & "=" & chr(39) & window.event.srcelement.value & chr(39)
end if

'IMPLEMENT THE FILTERING COMMAND STRING VIA THE RECORDSET COMMAND 'FILTER'.
rs.filter=filterval
```

XML FORMATS

As briefly mentioned earlier, the type of XML document written by ADODB isn't the same kind that SAS easily reads via its XML Libname engine. As a result, it's necessary to convert the resultant document into a generic XML format that will cooperate more readily. This can be accomplished using the MSXML2.DOMDocument.5.0 object (again, freely available on your Windows machine) to write out the appropriate content to a separate file using a series of 'nodes' populated by every field for every record of the already existing ADODB recordset. Here's the specific code that can be used to write any kind of database output as 'well-formed' XML:

```
Set xmldoc = createobject("Msxml2.DOMDocument.5.0")
Set objPI = xmldoc.createProcessingInstruction("xml", "version=""1.0""")
xmldoc.appendChild objPI

Set rootElement = xmldoc.createElement("catalog")
Set xmldoc.documentElement = rootElement

'START AT THE BEGINNING OF THE RECORDSET
rs.movefirst()

'LOOP THROUGH EACH RECORD ON THE RECORDSET.
do until(rs.eof)

   'CREATE AN ARBITRARY NAME FOR THE PARENT NODE (THIS WILL BE THE SAS DATASET NAME).
   set rowelement=xmldoc.createElement("tfls")
   rootElement.appendChild rowelement

   'COUNT THE NUMBER OF FIELDS/VARIABLES ON THE RECORDSET.
   numf=rs.fields.count-1

   'LOOP THROUGH EACH VARIABLE
   for i=0 to numf

      'GET THE VARIABLE NAME AND ASSOCIATED VALUE.
      fieldname=rs.fields(i).name
      fieldval=rs.fields(i).value

      'DIFFERENTIATE HANDING OF CHARACTER AND NUMERIC FIELDS.
      if instr(fieldname,"$")=0 then
         Set aElement = xmldoc.createElement(fieldname)
         if isNumeric(fieldval)=true then
            aElement.nodeTypedValue = fieldval
         elseif isNull(fieldval)=false then
            aElement.nodeTypedValue = cstr(fieldval)
         end if
         'APPEND THE FIELD AND VALUE ON AS A NEW 'NODE'.
         rowElement.appendChild aElement
      end if
   next
```

```
   'GO TO THE NEXT RECORD UNTIL FINISHED.
   rs.movenext
loop

'SAVE THE FINAL DOCUMENT IN THE APPROPRIATE FORMAT.
xmlDoc.save "C:\SAShappy_XML.xml"
```

It may seem a bit tedious, but it's a fairly convenient and reliable way to write out the necessary data into a format recognizable by SAS's XML libname engine (and you only have to code it once!).

READING XML DATA INTO SAS

Although entire papers have been written (and will probably continue to be for quite a while) concerning SAS's XML parser, for our purposes we'll touch only briefly on its usage here at end. With SAS version 8, in order to take advantage of the interoperability afforded by XML, SAS introduced an experimental XML parser, which was improved substantially in SAS 9. SAS's XML parser can be invoked very simply and allows XML files to be read for all intents and purposes as though they were SAS datasets. A simple 'libname' statement, specifying 'xml' as the data type, creates a reference to a specific XML file, and then a specific node/table can be read directly into a SAS dataset using the typical two-level naming convention:

```
*** SET UP LIBNAME POINTED TO THE XML METADATA.;
libname tflspecs xml "C:\SAShappy_XML.xml";

data metadata;
   set tflspecs.tfls;
...
run;
```

CONCLUSION

Peripheral information associated with SAS output in a clinical trial setting deserves to be stored and managed external to the actual program itself for a variety of reasons (only a few of which have been highlighted above). Possibly the best way (so far) to do that easily and efficiently is as an XML document that can be accessed for use in SAS via SAS's XML Libname engine. Such a document can also be relatively easily manipulated outside of the confines of SAS using a variety of parsers available within windows scripting languages.

REFERENCE

Fairfield-Carter, Brian, Sherman, Tracy, and Hunt, Stephen (2005), 'Instant SAS® Applications With VBScript, Jscript, and dHTML', Proceedings of the 2005 SAS Users Group International Conference.

Fairfield-Carter, Brian and Hunt, Stephen (2005), 'Managing a SAS® Programming Environment using Data Access Pages and the Microsoft Office Data Source Control', Proceedings of the 2005 SAS Users Group International Conference.

Shoemaker, Jack and Nelson, Greg S Barns (2003), 'XML Primer for SAS® Programmers', Proceedings of the 2003 SAS Users Group International Conference.

ACKNOWLEDGMENTS

We would like to thank Cara, PJ, Nicholas, Cole, our friends in Biostatistics and Programming at ICON Clinical Research (especially Syamala Ponnapalli and Jackie Lane), and Tim Williams for their consistent inspiration, encouragement and support.

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the authors at:

Stephen Hunt & Brian Fairfield-Carter
ICON Clinical Research
Suite 400, 555 Twin Dolphin Rd.
Redwood City, CA 94065
Email: hunts@iconus.com
Email: fairfieldcarterb@iconus.com