Paper 193-28

XML Primer for SAS® Programmers

Jack N Shoemaker, ThotWave Technologies, Cary, NC
Greg S Barnes Nelson, ThotWave Technologies, Cary, NC

ABSTRACT

XML is an SGML variant, which is an excellent messaging mechanism for passing data back and forth among applications. In this workshop, we discuss XML in the context of SAS applications and how XML can be used in the preparation and presentation of data. We explore some of the features of XML that make it a good partner for applications that are based on SAS. This workshop focuses on the tools that SAS provides for parsing XML documents so that the full power of SAS can be brought to bear on the manipulation and analysis of XML-encoded data. After a brief introduction to XML syntax, the workshop will demonstrate how to use the ever evolving XML library engine to make XML data available to SAS. Particular attention will be paid to the XMLMAP option, which provides the programmer with great control over how a particular XML document will appear to SAS in terms of the number of tables and units of observation of those tables. The intended audience for this workshop is intermediate SAS programmers with solid Base SAS skills and familiarity with markup languages such as HTML.

SCOPE OF WORKSHOP

As noted in the abstract above, the purpose of this hands-on workshop is to demonstrate how to use SAS to read and create XML documents. We will devote little time to the important topic of why you would want to read or create an XML document, or how XML documents might fit into your new or existing applications. That is not to say that these are not important topics. Rather, they are beyond the scope of this workshop. We encourage interested readers to review two excellent papers on XLM and SAS recently published and presented at SUGI27. The papers are “XML and SAS®: An Advanced Tutorial” by Greg Barnes Nelson and “<XML> at SAS - A More Capable XML Libname Engine” by Tony Friebel.

EXERCISES

This workshop will consist of a series of exercises which will demonstrate how to use SAS to read and create XML documents.

EXERCISE 1: A SIMPLE DOCUMENT

Viewed as any other file, an XML document is nothing more than an ASCII file. That is, it contains no special, or hidden, characters or word-processing paraphernalia. As a result you can use any text editor capable of creating an ASCII file to create an XML document. The choice of an appropriate editor is a personal decision on well beyond the scope of this presentation. Since this workshop occurs at SUGI, we will use the SAS Display Manager as our editor for this workshop. Bear in mind that any ASCII editor will do – UltraEdit, Kedit, Emacs, even Notepad.

To begin, let’s create a simple XML document. Type the following into the Editor window of Display Manager.

```
<?xml version="1.0"?>
<message>
Problems solved.  World saved!
</message>
```

Select File->Save As from the File menu. Type `exercise1.xml` in the Filename box and click the Save button. Congratulations! You have just used the SAS Display Manager editor to create a non-SAS file. Namely, an XML document called `exercise1.xml`. You can view this XML document either in the Editor window of Display Manager (it should be there already) or by using MS Internet Explorer. By default, Internet Explorer will render an XML document in a tree view. You should give this a try. Open up a Windows Explorer window and navigate to the folder containing exercise1.xml. Depending on the version of the Windows operating system that you are using the location of this document will vary. You may wish to use the Find or Search function to locate the document. In any event, once you have located the document, double-click the icon to render the document inside Internet Explorer. You should see something similar to this

<?xml version="1.0" ?>
<message>**Problems solved. World saved!**</message>

You’ve just learned about two ways to view the contents of an XML document. You can use the SAS Display Manager to view and modify the contents of the document by opening the document in the editor buffer. Or, you can use Internet Explorer to render a read-only tree view of the document. Although the tree view of this document looks very similar to the code view of the document, we will see shortly that the tree view is a neat way to view more complex XML documents.

EXERCISE 2: USING A DATA STEP TO CREATE AN XML DOCUMENT

We created the XML document in exercise 1 without any explanation about syntax or rules. That’s because the rules are very simple and flexible. First, every XML document must begin with the special header “`<?xml version="1.0"?>`”. There are other name-value attribute pairs like version= that you may specify. And, as the XML standard evolves and matures, more of these attributes may come into common usage. Notwithstanding, the bare minimum is what we have shown and will likely suffice for your applications for some time to come. The only other requirement for an XML document is that it contains balanced element tags. The names of these element tags are for you to choose. In our first exercise, we created an XML document with a single element called “message”. The contents of the message element is everything between the opening “<message>” and the closing “</message”. Note that every element must be properly balanced in this fashion. That is, and opening “<name>” must be closed by a closing “</name>”. At a very basic level, that’s really all there is to an XML document. As noted in the scope of workshop section above, there is plenty to consider when creating real-life XML documents. Also, just like HTML, XML elements may have numerous name-value attributes; however, we don’t intend to provide a class on XML at this workshop. Through the exercises, you should be able to pick up a few others rules about structure and usage, though that insight will be tangential to the thrust of this workshop.

Since an XML document is just a plain ASCII file, you can use DATA _NULL_ processing to create an XML document using PUT statements just as if you were creating a formatted report in the days before PROC QPRINT and PROC REPORT. For example, let’s say you have a SAS data set called EXERCISE2 which has two observations and one field called MESSAGE. You can create an XML document with two elements – one for each observation – by performing the following steps. First, write out the XML header. Second, create a top-level collection for the messages, called, say, “<messages>”. Then for each observation in EXERCISE2, write out “<message>” on a new line followed by the contents of the MESSAGE field from the EXERCISE2 data set followed by “</message>”. Finally, conclude the document with a closing “</messages>”. Your data step would look something like this.

```
data _null_;
  set exercise2 end = lastrec;
  file 'exercise2.xml';
  if _n_ = 1 then do;
    put '<?xml version="1.0"?>';
     put "<messages>";
  end;
  put "<message>" message "</message>";
  if lastrec then put "</messages>";
  run;
```

It is left to the reader to create the EXERCISE2 data set. At the hands-on-workshop, a data set will be provided for you. In any event, once you run (submit) this program, there will be a new XML document called `exercise2.xml` in the same folder where you found `exercise1.xml`. You should use Internet Explore to render this document. You should notice a minus sign next to the opening <messages> element. Move your mouse over this minus sign and left-click once. The minus sign will turn into a plus sign and the contained message elements will disappear. This is what is meant by a tree view. Internet Explorer allows you to collapse and expand elements as you wish. As a result, you have a great deal of control over how much detail you wish to view of a particular XML document.

EXERCISE 3: CONVERTING A SAS DATA SET TO XML

What could be more natural than to turn a SAS data set into an XML document? Based on the example above, you might be tempted to use a data step to encapsulate and write out all the fields in a particular data set. You could do that; however, SAS provides a much easier way of accomplishing this task through the XML LIBNAME engine. That is, you can create a SAS LIBNAME using the XML engine and then just write a data set to that aggregate location. Let’s use the data set called SASHELP.AIR, which is one of the sample data sets shipped with SAS, to create an XML document called EXERCISE3. Here are the steps to follow. First, define a LIBNAME called XMLOUT using the XML LIBNAME engine. Associate this LIBNAME with a file. Use a simple data step to create a new data set called XMLOUT.EXERCISE3 from SASHELP.AIR. Your data step should look something like this.

```
libname xmlout xml 'exercise3.xml'; run;

data xmlout.exercise3;
  set sashelp.air;
  run;
```

After you run this program, there should be a new XML document called erercise3.xml in the same folder as the previous XML documents. It is worth noting a few details about this small program and the resulting XML document. First, note that the LIBNAME statement refers to a specific file, exercise3.xml, and not a directory as is the normal custom. Admittedly, this is a bit confusing at first. Normally, FILENAME statements reference specific files and LIBNAME statements reference aggregate storage locations like directories. When using the XML LIBNAME engine, this distinction blurs. At an abstract level, the XML document is just a special aggregate storage location. It may be easier just to remember that the XML LIBNAME engine requires that you specify an exact file name, not just a directory location.

Once the XMLOUT LIBNAME is defined, you can use it in SAS programming just as if it were any other LIBNAME. The same rules for naming the LIBNAME hold. That is, the LIBNAME must be eight characters or less and may not start with a number, nor may the LIBNAME contain any characters other than numbers, the underscore, and letters. (Sorry about those name-space restrictions. The authors are just SAS users like you.) There is nothing special about the XMLOUT name. We could have used SASAVE, MYXML, or whatever LIBNAME flatters your prejudices.

If you have not already done so, you should use Internet Explorer to view the new XML document called EXERCISE3.XML. Note that the top-level container is called “<TABLE>”. This is name which SAS decided to use when building the XML LIBNAME engine. It is a sound and reasonable choice, though one made by the developers at SAS®. That is, there is no other special meaning to this element name. The <TABLE> collection contains a number of <ERERCISE3> elements. This name comes from the name of the data set we create with the data step in this example. That is, had we decided to create a data set called XMLOUT.MYAIR, then the <TABLE> collection would contain a set of <MYAIR> elements.

Inside the <EXERCISE3> elements there are two elements called <DATE> and <AIR>. These two elements correspond to the two fields in SASHELP.AIR called DATE and AIR. Finally, there is one <EXERCISE> element for each observation in SASHELP.AIR.

As you can see, it is remarkably easy to turn a SAS data set into an XML document. Perhaps it is too easy. Consider the overhead in terms of storage. Each field-value is encapsulated with over twenty characters of XML wrapper. For a small data set like SASHELP.AIR, that doesn’t amount to much. Surely, you would not want to convert your multi-million row trade-history table to an XML document. The added storage overhead would be prohibitive. On the other hand, many data-driven applications have parametric and meta data stored in smallish SAS data sets and tables. These smaller tables might be excellent candidates for conversion to XML.

EXERCISE 4: READING AN XML DOCUMENT – SIMPLE CASE

You may also use the XML engine to define a LIBNAME which is used as input. For example, now that we have an XML document called EXERCISE3.XML which contains rectangular data, we can create a SAS data step by reversing the process from exercise 3. That is, rather than use the XMLOUT LIBNAME on the DATA statement, we can use it on the SET statement instead to create a new SAS data set, say, EXERCISE4, in the SAS WORK area. For sake of clarity, let’s change the name of the LIBNAME from XMLOUT to XMLIN. It will still refer to the same XML document, EXERCISE3.XML. Your data step should look something like this.

```
libname xmlin xml 'exercise3.xml'; run;

data exercise4;
  set xmlin.exercise3;
  run;
```

After you run this program, you should find a data set called EXERCISE4 in the WORK library. Using the Explorer window inside SAS, find the WORK library and then double-click on the data set called EXERCISE4. This should render the data set in the VIEWTABLE viewer. SAS has no trouble parsing a regular and rectangular XML document such as EXERCISE3.XML. A later exercise will demonstrate how to use SAS to parse more complex and non-rectangular XML documents.

EXERCISE 5: CREATING AN XML DOCUMENT FROM SAS OUTPUT

The Output Delivery System (ODS) de-couples the content of procedure output from its presentation. You may think that when you run PROC MEANS on a data set that the procedure writes the results to the output window, or listing file, and that's it. Prior to the advent of ODS, this was the case. Now all SAS procedures write the contents of the procedure output to an ODS data store. ODS then uses a template to render these data to an output destination. The default output destination is the familiar output window, or listing file. Other output destinations include HTML, RTF, and, of course, XML.

To see how this works, let's render the output from PROC MEANS in an XML document. We will use the SASHELP.AIR data set again. As mentioned previously, the default output destination, called LISTING, is opened by default when you run SAS. To render the output to a different destination you encapsulate the procedure output in a pair of ODS statements. The first ODS statement opens the output destination and begins capturing the output. The second ODS statement closes the output destination and writes out the specified file. A full discussion of the Output Delivery System is beyond the scope of this workshop. Suffice it to say that the simple form of an ODS statement to open an XML destination is

```
ods xml file = "filename.xml"; run;
```

The corresponding ODS statement to close the XML destination is

```
ods xml close; run;
```

If we put a PROC MEANS statement inside these two ODS statements, the output from PROC MEANS will go to the file specified in the opening ODS statement. Your complete program should look something like this.

```
ods xml file = 'exercise5.xml'; run;

proc means data = sashelp.air;
  var air;
  run;

ods xml close; run;
```

If you haven't done so already, you should open EXERCISE5.XML using Internet Explorer. Try to find the actual values for the N, Mean, Std Dev, Minimum, and Maximum statistics. They should appear towards the bottom of the document inside the "<output-body>" element. This XML document is a far cry from the trivial example we started with in exercise 1. At this point you should have some appreciation of the wonderfully complex data structures that may be stored using XML. We will conclude this workshop with a final exercise to parse a non-rectangular XML document.

EXERCISE 6: PARSING NON-RECTANGULAR XML DOCUMENTS

The XML LIBNAME engine which ships with SAS 8.2 or earlier will not work with the following exercise. In order to use the LIBNAME options referenced in this example, you must first download and install the XMLMAP enhancements from the SAS web site. The URL for this update as of late 2002 is shown below.

http://www.sas.com/apps/demosdownloads/xmlengine_EXP_sysdep.jsp?packageID=000198

If you don't find anything at this location, start by navigating to the download section of the SAS web site and then look for Base SAS. This link contains both the download and instructions on how to apply this update to your version of SAS. We will not go into a discussion on how to apply this upgrade as part of this workshop. The workstations used for this HOW will already have this upgrade in place.

Consider the following problem. You wish to identify patients having a particular disease state by applying pattern-matching algorithms to a claims-history table. These patterns consist of one, or more diagnosis codes, and one or more procedure codes. You can encapsulate these data in an XML document like the one shown below.

```
<?xml version="1.0" ?>

<community>
  <disease>
     <Name>Aaa Bee Cee</Name>
     <Abbrev>ABC</Abbrev>
     <Events>
        <DX effdate="01JAN2001">123.45</DX>
        <DX effdate="01JAN2001">678.90</DX>
        <DX effdate="01JAN2001">345.67</DX>
        <PX effdate="01JAN2001">12345</PX>
        <PX effdate="01JAN2001">67890</PX>
     </Events>
  </disease>
  <disease>
     <Name>Dee Eee Eff</Name>
     <Abbrev>DEF</Abbrev>
     <Events>
        <DX effdate="01JAN2001">990.12</DX>
        <PX effdate="01JAN2001">12345</PX>
        <PX effdate="01JAN2001">67890</PX>
        <PX effdate="01JAN2001">34512</PX>
     </Events>
  </disease>
</community>
```

There are two disease states represented, ABC and DEF. Disease ABC has three diagnosis codes (DX) and two procedure codes (PX). Disease DEF has one diagnosis code and three procedure codes. These meta data are clearly non-rectangular and would not fit easily into a single SAS data set.

The first challenge is to determine how to store these data in rectangular SAS tables. For the sake of discussion, let's agree that one sensible solution would be to create one table of disease-specific information; one table of DX events, and one table of PX events. We wish to be able to link the DX and PX event tables to the disease table, so certain key information must be retained in the DX and PX event tables.

The XMLMAP= option on the LIBNAME statement allows you to specify an XML file that defines how to parse an XML document into various bits. As noted above, the XMLMAP= option will only work if you have applied the patch described above. The XMLMAP= options changes the way that the XML LIBNAME engine parses the XML document. The revised syntax looks like this.

```
libname xmlin
  xml 'SUGI28.HOW.xml'
  xmlmap = 'SUGI28.HOW.map'; run;
```

That is, the XMLMAP= option refers to a special XML document which contains information that the XML LIBNAME engine uses to parse the XML document. We'll examine the structure of the XMLMAP file in a moment. For now, let's stipulate that it will parse the XML document above into three SAS data sets called DISEASE, DXEVENTS, and PXEVENTS. Using the LIBNAME statement above, you can print these "data sets" directly as follows.

```
proc print data = xmlin.disease;
proc print data = xmlin.dxevents;
proc print data = xmlin.pxevents;
```

Obviously, the “programming” occurs in the XMLMAP file. The remainder of this workshop will discuss the elements of the XMLMAP file and how to construct one for the XML document listed above.

EXERCISE 7: BUILDING AN XMLMAP DOCUMENT

The XMLMAP document is itself an XML document. The content and structure of the XMLMAP document are defined by SAS. Like all XML documents, the XMLMAP file must begin with the special XML header. The top-level collection name for the XMLMAP document is <SXLEMap>. So, the basis outline for an XMLMAP document is

```
<?xml version="1.0" ?>

<SXLEMap>
</SXLEMap>
```

Data sets, or tables are specified by <TABLE> elements. We wish to parse our XML document into three tables, so it stands to reason, that the SXLEMap collection will contain three <TABLE> elements as follows.

```
<?xml version="1.0" ?>

<SXLEMap>
  <TABLE name=”DISEASE”>
  </TABLE>
  <TABLE name=”DXEVENTS”>
  </TABLE>
  <TABLE name=”PXEVENTS”>
  </TABLE>
</SXLEMap>
```

Next we need to tell SAS where in the XML document the tables begin and end. This is done with the <TABLE_XPATH> element which employs standard XPATH syntax to specify where in the XML document tree a table begins. For the DISEASE table, this would be “/community/disease”. That is, the <disease> elements in the XML document will define rows, or observations, in the resulting DISEASE table. The DISEASE table element would therefore look like this.

```
<TABLE name="DISEASE">
   <TABLE_XPATH> /community/disease
</TABLE_XPATH>
</TABLE>
```

Next we need to specify which elements inside the <disease> container correspond to columns in the resulting SAS DISEASE data set. This is done with the <COLUMN> element which also uses XPATH syntax to locate where the values for the column may be found. In out example, the SAS data set DISEASE will contain the name of the disease as well as the short name or abbreviation. The name is found at /community/disease/Name and the short name is found at /community/disease/Abbrev. In addition to specifying the location of these elements in the document tree, the <COLUMN> element also contains sub-elements which you can use to specify the type, length, label, format, and informat of the column contents. Here is what the <COLUMN> element looks like for the Name column in the DISEASE data set.

```
<TABLE name="DISEASE">
   <TABLE_XPATH> /community/disease
</TABLE_XPATH>
    <COLUMN name="Name">
         <XPATH> /community/disease/Name
</XPATH>
         <TYPE> character </TYPE>
         <DATATYPE> string </DATATYPE>
         <LENGTH> 40 </LENGTH>
         <LABEL> Formal Disease Name </LABEL>
      </COLUMN>
</TABLE>
```

Note the two elements called <TYPE> and <DATATYPE>. The element called <TYPE> is the resulting SAS data type. The element called <DATATYPE> is the XML data type and only comes in flavors of string and number. Since you can use INFORMATS to read-in values, you can safely treat almost all XML data as string. Our XML document associates effective dates with the various DX and PX values. We would like to store this information as a column in the resulting DXEVENTS and PXEVENTS tables. We use the XPATH ‘@’ syntax to access these attribute values. Since these values are actually dates, we prefer to store then as SAS dates. We can use the <FORMAT> and <INFORAMT> sub-elements to accomplish this. For example, the <COLUMN> element for effective dates might look like this.

```
<TABLE name="DXEVENTS">
   <TABLE_XPATH> /community/disease/Events/DX
</TABLE_XPATH>
   <COLUMN name="EffDate">
      <XPATH>
/community/disease/Events/DX@effdate </XPATH>
      <TYPE> numeric </TYPE>
      <DATATYPE> string </DATATYPE>
      <LENGTH> 9 </LENGTH>
      <LABEL> Effective Date </LABEL>
      <FORMAT width = "9"> date </FORMAT>
      <INFORMAT width = "9"> date </INFORMAT>
   </COLUMN>
</TABLE>
```

We mentioned at the outset of this exercise, that we would retain certain key information in DXEVENTS and PXEVENTS so that we could later link DXEVENTS and DISEASE. This is done by specifying the attribute retain=”YES” on the column element. For example the column element inside the DXEVENTS table for the short name might begin like this.

```
<COLUMN name="ShortName" retain="YES" >
```

You now have most of the basic elements required to construct an XMLMAP document. There are a number of other elements not mentioned here which provide even greater control over parsing. Those may be found in the XMLMAP documentation found on the SAS web site. Without looking ahead, see if you can build an XMLMAP document for our XML document by following the following rules. First, the top-level container is called <SXLEMap>. Second, tables are specified by <TABLE> elements. The <TABLE> elements contain a sub-element to indicate where an observation for the table begins, <TABLE_XPATH> as well as <COLUMN> elements for each desired output column. The <COLUMN>, in turn, contains sub-elements to specify the location of the column in the XML documents, <XPATH>, as well as various attributes on the column such as LENGTH, TYPE, FORAMT, and INFORMAT.

Here is our solution to the full XMLMAP file for this exercise.

```
<?xml version="1.0" ?>

<SXLEMap>

   <!-- TABLE (DISEASE)
-->
   <!-- top level disease description
-->
   <TABLE name="DISEASE">
```

```
      <TABLE_XPATH> /community/disease
</TABLE_XPATH>
      <TABLE_LABEL> Individual disease
communities </TABLE_LABEL>

      <!-- Name
-->
      <COLUMN name="Name">
         <XPATH> /community/disease/Name
</XPATH>
         <TYPE> character </TYPE>
         <DATATYPE> string </DATATYPE>
         <LENGTH> 40 </LENGTH>
         <LABEL> Formal Disease Name </LABEL>
      </COLUMN>

      <!-- Short Name
-->
      <COLUMN name="ShortName">
         <XPATH> /community/disease/Abbrev
</XPATH>
         <TYPE> character </TYPE>
         <DATATYPE> string </DATATYPE>
         <LENGTH> 10 </LENGTH>
         <LABEL> Short Disease Name </LABEL>
      </COLUMN>

   </TABLE>

   <!-- TABLE (DXEVENTS)
-->
   <!-- List of DX Events
-->
   <TABLE name="DXEVENTS">
      <TABLE_XPATH>
/community/disease/Events/DX </TABLE_XPATH>
      <TABLE_LABEL> DX Events </TABLE_LABEL>

      <!-- Short Name
-->
      <COLUMN name="ShortName" retain="YES" >
         <XPATH> /community/disease/Abbrev
</XPATH>
         <TYPE> character </TYPE>
         <DATATYPE> string </DATATYPE>
         <LENGTH> 10 </LENGTH>
         <LABEL> Short Disease Name </LABEL>
      </COLUMN>

      <!-- DX Code
-->
      <COLUMN name="DX">
         <XPATH> /community/disease/Events/DX
</XPATH>
         <TYPE> character </TYPE>
         <DATATYPE> string </DATATYPE>
         <LENGTH> 6 </LENGTH>
         <LABEL> ICD9CM DX Code </LABEL>
      </COLUMN>

      <!-- Effective Date
-->
      <COLUMN name="EffDate">
         <XPATH>
/community/disease/Events/DX@effdate </XPATH>
         <TYPE> numeric </TYPE>
         <DATATYPE> string </DATATYPE>
         <LENGTH> 9 </LENGTH>
         <LABEL> Effective Date </LABEL>
         <FORMAT width = "9"> date </FORMAT>
         <INFORMAT width = "9"> date
</INFORMAT>
      </COLUMN>

   </TABLE>

   <!-- TABLE (PXEVENTS)
-->
   <!-- List of PX Events
-->
   <TABLE name="PXEVENTS">
      <TABLE_XPATH>
/community/disease/Events/PX </TABLE_XPATH>
      <TABLE_LABEL> PX Events </TABLE_LABEL>

      <!-- Short Name
-->
      <COLUMN name="ShortName" retain="YES" >
         <XPATH> /community/disease/Abbrev
</XPATH>
         <TYPE> character </TYPE>
         <DATATYPE> string </DATATYPE>
         <LENGTH> 10 </LENGTH>
         <LABEL> Short Disease Name </LABEL>
      </COLUMN>

      <!-- PX Code
-->
      <COLUMN name="PX">
         <XPATH> /community/disease/Events/PX
</XPATH>
         <TYPE> character </TYPE>
         <DATATYPE> string </DATATYPE>
         <LENGTH> 5 </LENGTH>
         <LABEL> CPT-4 PX Code </LABEL>
      </COLUMN>

      <!-- Effective Date
-->
      <COLUMN name="EffDate">
         <XPATH>
/community/disease/Events/PX@effdate </XPATH>
         <TYPE> numeric </TYPE>
         <DATATYPE> string </DATATYPE>
         <LENGTH> 9 </LENGTH>
         <LABEL> Effective Date </LABEL>
         <FORMAT width = "9"> date </FORMAT>
         <INFORMAT width = "9"> date
</INFORMAT>
      </COLUMN>

   </TABLE>

</SXLEMap>
```

CONCLUSION

We hope that this workshop has whetted your appetite to do more with XML and SAS. XML is an excellent choice for storing messages and meta data. If you build data-driven applications, you likely have some of these sorts of data already lying about. SAS provides some very clean and elegant interfaces to data in XML format which you can use to your advantage. The cocktail is intoxicating and the possibilities are boundless.

CONTACT INFORMATION

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

Jack N Shoemaker
ThotWave Technologies
Cary, NC
shoe@thotwave.com

Greg S Barnes Nelson

ThotWave Technologies
Cary, NC
greg@thotwave.com