Paper TU-17

PROC FORMAT in Action

Jack N Shoemaker, ThotWave Technologies, Cary, NC

ABSTRACT

The PROC FORMAT subsystem provides an efficient and compact way to store all sorts of facts and data for data-driven applications. This paper will focus on PROC FORMAT in action. That is, building user-defined formats from extant tables and using PROC FORMAT in conjunction with other SAS procedures like SUMMARY, FREQ, TABULATE, REPORT, and PRINT. Since these other SAS procedures are format-aware, judicious use of user-defined formats can save hours of coding and result in more robust, data-driven applications.

WHAT IS SAS®?

That may seem like an odd question to pose to a group of SAS® programmers and developers. But it is one that your author has encountered frequently over the years. SAS® is a programming language, or set of programming languages depending upon how you do the maths. There is the ubiquitous base language with its data steps and procs. There is the macro language that bears close resemblance to the base language, but doesn't follow quite all the same rules. There is the SAS® component language nee screen control language, or SCL, which doesn't look a thing like the base language save the use of the semicolon as a statement terminator. In fact, if SAS® stands for anything, it is "semicolon always semicolon."

The data-step portion of the base language would be recognized as a programming language to a developer, or programmer, of any other programming language. The data step provides syntax for assignment, looping, and logical branching that are basis of any language. The base language also supplies pre-built procedures, or procs, some of which would be recognized by users of reporting and statistical packages.

USING FORMATS IN SAS®

The SAS® system provides a clean mechanism for separating the internal value of a particular column from its display. This is done through formats that associate a particular display format with a column of data. SAS® supplies scores of formats to display numbers, character strings, and dates. For example, the $CHAR. format will display a character string preserving all leading and trailing spaces. The COMMA. format will display a number with comma punctuation. The DATE. format will display a SAS® serial date in familiar day-month-year notation.

Despite this wealth of formats, you can easily run into a situation where a format does not exist to suit your needs. For example, your data may contain a column called sex that has values of 1 and 2 representing females and males respectively. No SAS®-supplied format will de-reference these coded values for you. You could use a data step to create a new column to de-reference these coded values. For example,

```
data new;
  set old;
  length SexString $ 6;
  if sex = 1 then SexString = 'Female';
  else if sex = 2 then SexString = 'Male';
  run;
```

SAVING SPACE

There is nothing inherently wrong with this approach. If the existing data set called old has a numeric column called sex which has values of 1 or 2 only, this data step will create a new character column called SexString which will take on values of 'Female' or 'Male'. One problem with this approach might be a waste of computer resources, namely, storage. If the old data set is small, this likely isn't a concern. However, if the old data set were a twenty-million row claims history file from the administrative records of a health insurer, the creation of essentially a copy of the data set with a new six-character column may be a show stopper.

If a SEX. format existed which displayed 1's as 'Female' and 2's as 'Male', you could display the values in old directly without creating a whole new data set. For example, to count the numbers of males and females in the old data set you could use PROC FREQ as follows.

```
proc freq data = old;
  format sex sex.;
  table sex;
  run;
```

But the SEX. format doesn't come with SAS®. To make the above proc step work, you must create a user-defined format using PROC FORMAT. For example,

```
proc format;
   value sex
     1 = 'Female'
     2 = 'Male'
     ;
   run;
```

The code above will create a numeric format called SEX. The PROC FORMAT block contains three statements each terminating wit a semicolon. The first statement calls PROC FORMAT. The second statement is a value clause that defines the name and contents of the user-defined format. The final statement is the run statement which, though not technically required, is good programming practice nonetheless.

The PROC FORMAT block may contain as many value clauses as you desire. The value clause consists of a format name followed by a set of range-value pairs that define what the format does. Ranges may be single items as shown in the example above, or ranges of items separated by commas, or ranges expressed as from – through. Value clauses with format names that begin with a dollar sign define character formats; names without a dollar sign define numeric formats as shown above. Note that the distinction between numeric and character applies to column to which the format will be applied – not the displayed value. For example, the example above defines a numeric format because it will be applied to a numeric column. Once defined you may use the user-defined format called SEX. anywhere you would use a SAS®-supplied format. Note that all formats, user-defined or otherwise, end with a period when referenced as in the PROC FREQ example above.

SEPARATING CODE AND DATA

Another defect with the data-step approach to our simple problem is that it's not terribly robust. Actual data are often noisy. Despite written documentation to the contrary, the sex column may contain missing values, or other unknown codes like 8 or 9. Or, the domain of allowable values may change over time. Perhaps 3 is an allowable value for indeterminate. You can account for all these conditions in the data step. However, this approach

requires that you change your code to account for these changes in the data. And, if the sex column is used at multiple points in your application, you may end up changing code in many places. For coded data only slightly more complicated that our sex example, this can quickly devolve into a maintenance nightmare. Using a user-defined format provides a more robust solution to this problem because it separates the "code" in the application – the PROC FREQ in our simple example – from the "data" – that is, the input data set and translation data which de-references the coded values of sex. For example, we can easily modify our initial definition of SEX. to include a translation for value 3 and a catch-all for all other values.

```
proc format;
   value sex
     1 = 'Female'
     2 = 'Male'
     3 = 'Indeterminate'
     other = 'Unknown'
     ;
   run;
```

We can now run our PROC FREQ "application" without change. Furthermore, we have surfaced the translation rules for the sex codes rather than burying them in an if-then-else tree in a data step. This brief introduction is not meant to be a full description of the FORMAT procedure. For that, you should turn to the PROC FORMAT section of the SAS® reference manual. Rather, this should whet your appetite about the possibilities. Unlike the familiar aspects of data-step programming or the reporting-package-like functionality of PROC PRINT and PROC REPORT, the FORMAT procedure has no close analog in other programming languages. It is not quite an enumerated data type. Nor, is it exactly like a dictionary table in Python. Though these constructs are close. In your author's humble opinion, PROC FORMAT is a sub-system of SAS® deserving of its own special treatment like SQL. What follows is a loosely-knit series of examples showing how to use PROC FORMAT in everyday applications. It is hoped that some of these examples will resonate and help you with your day-to-day programming tasks.

FORMATS ARE STORED IN SAS® CATALOGS

Broadly speaking, the SAS® system divides the world into two types of data objects: the data set and the catalog. Of course, the data step creates data sets. Many procedures have OUT= directives which also create data sets. Virtually everything else ends up in a catalog, for example, stored SCL code, and saved graphics output. The user-defined formats created by PROC FORMAT are no exception.

You refer to data sets with what is called a two-level name. For example, SASAVE.SESUG refers to a data set called SESUG in a library called SASAVE. Library names refer to aggregate storage locations in the file systems for your particular operating system. The association of library name to aggregate storage location is done through the LIBNAME statement. For example, the following statement would create a library called SASAVE.

```
libname sasave '/usr/data/sasave';
```

For modern operating systems like Unix, VMS, and Windows which support tree-structure directories, the aggregate storage locations are just directories or folders. Under older operating systems, like MVS, the aggregate storage locations refer to (confusingly) OS data sets that have been pre-allocated through magical incantations known as JCL. If you have never heard of the terms MVS, JCL, or DD, consider yourself fortunate to be so young.

Unlike data sets which contain only one object – the data set, catalogs may contain many items known as members. To refer to a catalog member, you use a four-level name. For example, SASAVE.SESUG.EXAMPLE.FORMATC refers to a catalog member called EXAMPLE in the catalog called SESUG in the library called SASAVE. The final node of this four-level name, FORMATC, means that EXAMPLE is a user-define character format.

If you are using one of the operating systems listed above which support tree-structured directories, you can browse the directory contents and see the actual file names which correspond to the data set and catalog listed above. For example, if you are running version 8 of the SAS® system under Windows NT, then the data set would have this name:

```
SESUG.sas7bdat
```

While the catalog would appear as:

```
SESUG.sas7bcat
```

The default format catalog is LIBRARY.FORMATS. That is, a catalog called FORMATS in the library called LIBRARY. The library called LIBRARY should be created by the person, or group, who administers SAS® at your site. The installation process does not create this library. However, somewhat paradoxically, SAS® searches for a library called LIBRARY for many of its default operations, like locating user-defined formats. The definition for the library called LIBRARY usually occurs in your AUTOEXEC.SAS file that you should find in the SAS® root directory that contains the SAS® executable file, sas.exe.

You can use PROC CATALOG to list the contents of a format catalog or any other SAS® catalog for that matter. For example, the following code fragment will display a list of all the members of the default catalog, LIBRARY.FORMATS:

```
proc catalog c = library.formats;
  contents stat;
  run;
```

The output will look something like this:

```
#  Name    Type       Description
-----------------------------------
1  AGE     FORMAT
2  PHONE   FORMAT
3  AGE     FORMATC
4  MYDATE  INFMT
```

The actual display will be wider than what's shown here which has been truncated to fit within the margins of this paper. Note that there are three different member types: FORMAT, FORMATC, and INFMT. The FORMAT member type specifies a numeric or picture format. The FORMATC format specifies a character format. And the INFMT member type specifies an informat that is used to read rather than display data.

In version 8, the description attribute is left blank. In earlier versions, the description attribute contains some technical details about the format like default length and maximum size. In any event, you should use the description attribute to provide short documentation about the user-defined format. The name-space for user-defined formats still remains just eight characters which means that your format names will look pretty dense, like variable names and such in the pre-version 7 days. The description attribute provides a simple way to compensate for this lingering restriction.

The following code fragment uses PROC CATALOG to modify the description attribute of two members of the temporary catalog WORK.FORMATS.

```
proc catalog c = work.formats;
  modify
    age.format( description = 'Age Map' );
  modify
    age.formatc( description = 'Age Decoder'
);
  run;
```

If your SAS® system administrators have acted in a responsible fashion, you will not be allowed to modify the common LIBRARY.FORMATS catalog. So, the example above uses the temporary format catalog called WORK.FORMATS that is created in the temporary WORK library. Just as data sets created in the WORK library disappear at the end of your SAS® session, a format catalog created in the WORK library will also disappear. Notwithstanding, for the purposes of illustration and discussion the remainder of this paper will use the temporary WORK library.

The resulting display from PROC CATLOG would look like this:

```
#  Name     Type        Description
-----------------------------------
1  AGE      FORMAT      Age Map
2  PHONE    FORMAT
3  AGE      FORMATC     Age Decoder
4  MYDATE   INFMT
```

REVIEWING FORMAT DEFINITIONS

The preceding example shows how to list the members of a format catalog. You can also look at the contents of a particular user-defined format. One technique is to use the FMTLIB= option of PROC FORMAT. For example, the following code fragment will display the contents of the user-defined format called AGE.

```
proc format
  library = work.formats fmtlib;
  select age.;
  run;
```

A truncated version of the output of this code might look like this:

```
---------------------------------------
|       FORMAT NAME: AGE      LENGTH:
|   MIN LENGTH:   1  MAX LENGTH:  40  D
|-------------------------------------
|START           |END             |LABE
|----------------+----------------+----
|               0|              20|1
|              20<              30|2
|              30<HIGH            |3
```

The FMTLIB display shows the start and end values of the format range as well as the resulting label. In this example, the label is a single digit – 1, 2, or 3 – which presumably needs to be de-coded with a subsequent format definition. The less-than symbols (<) after 20 and 30 in the start column indicate that those values are not in the specified range. This matters for variables that take on continuous values. The label 1 is associated will all values between 0 and 20 including the end-point values 0 and 20. The label 2 is associated with all values between 20 and 30 not including the exact value of 20 which is in the first range. Similarly, the label 3 does not include the exact value 30, but does all other values above 30. This may represent more control over your data than you need. Notwithstanding, it's nice to know that you have this control should you need it.

TURNING A USER-DEFINED FORMAT INTO A SAS® DATA SET

The FMTLIB= option on PROC FORMAT provides a mechanism for displaying the contents of a user-defined format as regular SAS® output. You can also unload the contents of a user-defined format into a SAS® data set using the CNTLOUT= option on PROC FORMAT. For example, the following code fragment will create a data set called CNTLOUT from the all the user-defined formats stored in the catalog called WORK.FORMATS.

```
proc format library = work.formats
  cntlout = cntlout;
  run;
```

The resulting SAS® data set will contain the following twenty columns.

```
Variable    Type  Label
-----------------------------------------
DATATYPE    Char  Date/time/datetime?
DECSEP      Char  Decimal separator
DEFAULT     Num   Default length
DIG3SEP     Char  Three-digit separator
EEXCL       Char  End exclusion
END         Char  Ending value for format
FILL        Char  Fill character
FMTNAME     Char  Format name
FUZZ        Num   Fuzz value
HLO         Char  Additional information
LABEL       Char  Format value label
LANGUAGE    Char  Language for date strings
LENGTH      Num   Format length
MAX         Num   Maximum length
MIN         Num   Minimum length
MULT        Num   Multiplier
NOEDIT      Num   Is picture string noedit?
PREFIX      Char  Prefix characters
SEXCL       Char  Start exclusion
START       Char  Starting value for format
TYPE        Char  Type of format
```

If that seems like a lot of columns, it is. Most are there to provide the extra levels of control which are needed in specific circumstances. In fact there are only three required columns: FMTNAME, START, and LABEL. In addition to these required columns it is good habit to include the TYPE column which explicitly tells PROC FORMAT that you are building a numeric or character format. Of course if your format is to include ranges, you will need to include an END column as well as the START column. Finally, the HIGH, LOW, and OTHER keywords are coded in the HLO column. In summary, the six commonly useful columns are listed below:

```
Variable    Type  Label
-----------------------------------------
FMTNAME     Char  Format name
TYPE        Char  Type of format
START       Char  Starting value for format
END         Char  Ending value for format
LABEL       Char  Format value label
HLO         Char  Additional information
```

Here's what the CNTLOUT data set for the AGE format looks like:

```
FMTNAME TYPE START  END  LABEL HLO

   AGE    N     0    20     1
   AGE    N    20    30     2
   AGE    N    30    HIGH   3    H
```

USER-DEFINED FORMATS AS TABLE LOOK UPS

You can use user-defined formats to display or write-out coded values in raw data. For example, the values of 'M' and 'F' could become 'Male' and 'Female' if displayed using a user-defined format called $SEX. In a sense, the user-defined format called $SEX. is just a two-column lookup table with 'M' and 'F' as the key values and 'Male' and 'Female' as the looked-up return values. You can use user-defined formats in just this fashion in a data step by using the PUT() function. Following along our example, if you wish to create a new data-step variable called 'description' from an existing data-step variable called 'sex' using a user-defined format called $SEX., you could use a piece of code like this:

```
description = put( sex, $sex. );
```

This technique allows you to re-write if-then-else trees and replace then with a single line of code. For example, assume that you have a set of discount factors stored in a user-defined format called $DISC.

```
proc format;
  value $disc
    'ABC' = 0.20
    'DEF' = 0.25
    'XYZ' = 0.00
    other = 0.00;
```

You could replace code that looks like this:

```
if vendor = 'ABC' then discount = 0.20;
else if vendor = 'DEF' then discount = 0.25;
else if vendor = 'XYZ' then discount = 0.00;
```

With a single statement that looks like this:

```
discount = put( vendor, $disc. );
```

This technique also has the added advantage of separating the data – the table of discount factors – from the code. If you need to add or change the discount values for your vendors, you simply change that data outside of the data step and leave your existing data-step code alone.

One word of caution: the PUT() function always returns a character string. So, if you mean to use the return value as a number you must take some action to cause SAS® to convert the character string to a number. For example:

```
length discount 8;
discount = put( vendor, $disc. );
```

or

```
net = gross * ( 1 - put( vendor, $disc. ) );
```

That is, either explicitly declare the return variable as a number. Or, perform some sort of arithmetic on the result inside the assignment statement.

A simpler example still is to create an user-defined informat instead of a format and use the input() function instead of the put() function. For example:

```
proc format;
  invalue disc
    'ABC' = 0.20
    'DEF' = 0.25
    'XYZ' = 0.00
    other = 0.00;
```

```
discount = input( vendor, disc. );
```

This final technique has the added advantage of not producing and conversion messages in the SAS log. You may consider these messages harmless when you expect to see them. On the other hand, if you consider any conversion message in the SAS log to be a sign of sloppy or suspect programming, you should use a user-defined informat in conjunction with the input() function.

CREATING A USER-DEFINED FORMAT FROM A DATA SET OR TABLE

You may also create a user-defined format from an existing data set or data-base table. Imagine that your vendor discount table has hundreds or thousands of entries. Manually coding this many entries in a value clause would be both error-prone and time-consuming. Fortunately PROC FORMAT provides an analog to the CNTLOUT= option called CNTLIN= which loads a user-defined format from a data set. The only requirement is that the field names on the data set specified by the CNTLIN= option must conform to the list of field names listed in the discussion about the CNTLOUT data set above.

For example, consider an existing data set called DISCOUNT with two columns called VENDOR and DISCOUNT. You could build a suitable CNTLIN= data set from the DISCOUNT data set as follows:

```
data cntlin(
  keep = fmtname type hlo start label );
  retain fmtname 'disc' type 'C';
  set discount end = lastrec;
  start = vendor; label = put( discount, 6.2
);
  output;
  if lastrec then do;
    hlo = 'O'; label = '0.00';
    output;
  end;
  run;
```

Note that the CNTLIN data set has only five columns. Actually, only three are required – FMTNAME, START, and LABEL. As a matter of good habit, including the TYPE column with values of 'C' for character and 'N' for numeric is strongly advised. Also, since our example includes the use of the HIGH keyword, we must include the HLO column as well.

The following code fragment will create the user-defined format called $DISC. In the temporary format catalog in the WORK library.

```
proc format cntlin = cntlin; run;
```

If you wish to store this format to a permanent library, like LIBRARY, you need to include the LIBRARY= option as well. For example,

```
proc format
  cntlin = cntlin library = library; run;
```

Building user-defined formats using CNTLIN data sets also allows you to build self-modifying formats. For example, consider the need to build a format with values of 'This Month' for the current month, 'Last Month' for the previous month, and 'Really Old' for dates prior to that. Obviously as time marches on, you need to update the dates associated with these ranges. Here's how you could accomplish this feat using a CNTLIN data set with three observations.

```
data cntlin(
  keep = fmtname type hlo start end label );
  retain fmtname 'MyDate' type 'N';
  length label $ 10;
  rundate = today();
  start = intnx( 'month', rundate, 0 );
  end = intnx( 'month', rundate, 0, 'E' );
  label = 'This Month';
  output;
  start = intnx( 'month', rundate, -1 );
  end = intnx( 'month', rundate, -1, 'E' );
  label = 'Last Month';
  output;
  hlo = 'O';
  label = 'Really Old';
  output;
  stop;
  run;
```

PRODUCING HYBRID FORMATS

You can also define user-defined formats which combine, or use, other user-defined formats or SAS®-supplied formats. A common situation when this need arises occurs when handling date values which contain missing values. Suppose you have a column which contains a SAS® serial date most of the time. At other times it contains one of two special missing values .N or .Z. You would like to display .N and .Z with some notation, but otherwise use the SAS® DATE9. format to display the date values. The following code fragment will create a user-defined format called OTDATE which does just that.

```
proc format;
  value otdate
     .Z = 'Some Zs'
     .N = 'Some 9s'
    other = [date9.];
```

The trick is to encapsulate the embedded format in square brackets. On operating systems which do not support this character, you may replace '[' with '(|' and ']' with '|)'.

You can do the same thing when reading data. For example, assume that a date field in raw data either contains eight zeroes, eight nines, or a properly-formated date in YYYYMMDD format. Rather than read the field as a character string and convert it as necessary, you can create a user-defined informat to do the work for you. For example, the following code fragment will create a user-defined format called INDATE which reads the date field as described above.

```
proc format;
  invalue indate
     '00000000' = .Z
     '99999999' = .N
    other = [yymmdd8.];
```

To see how this all works together, consider the following short SAS® program which uses both the INDATE informat as well as the OTDATE format.

```
data SESUG;
  infile cards;
  input aDate indate8.;
  cards;
00000000
99999999
20000605
;
run;

proc print data = SESUG;
  format aDate otdate.;
```

The results look like this:

```
aDate

  Some Zs
  Some 9s
  05JUN2000
```

MULTI-VALUE LABELS

The next topic for this paper is multi-value labels. That is, how to handle situations where you want to use a user-defined format to associate more than one attribute with a given key value. For example, in our vendor example above, we might have a region and salesperson associated with each vendor as well as a discount amount.

There are two choices: create a separate user-defined format for each attribute, or create label which stores both attributes using some unique character to distinguish one attribute from the other.

Consider the following VENDOR data set

```
data vendor;
  infile cards;
  input vendor $ region $ salesp $;
  cards;
ABC NE Alice
DEF MW Molly
XYZ SE Linda
;
run;
```

The following code fragment will create a CNTLIN= data set which will create two separate user-defined formats – one for the region and one for the salesperson.

```
data cntlin( keep = fmtname type start label
);
  retain type 'C';
  set vendor;
  start = vendor;
  fmtname = 'region'; label = region; output;
  fmtname = 'salesp'; label = salesp; output;
  run;

proc sort data = cntlin; by fmtname; run;

proc format cntlin = cntlin; run;
```

We could have created two separate CNTLIN data sets and fed them to PROC FORMAT one at a time. Instead we created a CNTLIN data set which contains two output rows for each row of input from the VENDOR data set. When using the later technique the PROC SORT is crucial. Using it ensures that all the region definitions come first followed by all the salesperson definitions.

Alternatively, you could create a label which concatenates the region and salesperson values with a delimiting character like '#'. For example,

```
data cntlin( keep = fmtname type start label
);
  retain fmtname 'vinfo' type 'C';
  set vendor;
  start = vendor;
  label = region || '#' || salesp;
  run;
```

```
proc format cntlin = cntlin; run;
```

The $VINFO format is not very useful as a display format. It is designed for use inside a data step in conjunction with the PUT() function. For example, the following data-step code fragment will create two data-step variables called REGION and SALESP from VENDOR using the user-defined format $VINFO.

```
length region $ 2 salesp $ 5 vinfo $ 8;
vinfo = put( vendor, $vinfo. );
region = scan( vinfo, 1, '#' );
salesp = scan( vinfo, 2, '#' );
```

Choice of the delimiting character is crucial when using this technique. The character you chouse as a delimiter must never appear as in either of the tokens inside the concatenated label.

MULTI-VALUE RANGES

Starting with version 8, PROC FORMAT allows you to define multi-value ranges. That is, ranges which have overlapping values. Normally overlapping ranges cause the FORMAT procedure to terminate in error. The special "multilabel" option instructs PROC FORMAT to construct a user-defined format using overlapping ranges. You may ask yourself, "Why on earth would I want to do this?" An example will illustrate the point.

```
proc format;
  value age(multilabel)
    20 - 29 = '20 - 29'
    30 - 39 = '30 - 39'
    40 - 49 = '40 - 49'
    50 - 59 = '50 - 59'
    60 - high = '60 +++'
    20 - 35 = '20 - 35'
    36 - 55 = '36 - 55'
    55 - high = '55 +++'
    ;
  run;
```

The user-defined format called AGE. has overlapping ranges. The last three entries in the value clause overlap or are contained in the first five entries. To see how this might be useful, consider passing a data set containing income and age through PROC MEANS as follows:

```
proc means data = SESUG
  n min max maxdec = 0;
  class age / mlf;
  format age age.;
  var income;
  run;
```

Note the 'mlf' option on the class statement. This instructs PROC MEANS that a multi-label format is in use. The resulting output is, frankly, quite remarkable:

```
The MEANS Procedure

                Analysis Variable : Income

Age            N       Minimum         Maximum
-----------------------------------------------
20 - 29     2170             0          150000
20 - 35     3501             0          150000
30 - 39     2201         10000          160000
36 - 55     4417         10000          180000
40 - 49     2189         20000          180000
50 - 59     2289         30000          170000
55 +++      2286         30000          190000
60 +++      1151         40000          190000
-----------------------------------------------
```

Before the multilabel option, there was really no way to do this save some very convoluted data-step trickery. Now, it is as simple as a single user-defined format and a PROC MEANS.

CONCLUSION

The FORMAT procedure is a real gem. You may use it in a variety of situations to make your code more robust and easier to maintain. Also, the use of PROC FORMAT encourages separation of code and data which leads to cleaner and more understandable code. This paper has explicated a couple of common uses for user-defined formats. There are plenty more that await you in your applications. Would that this paper aid you in that process of discovery.

CONTACT INFORMATION

Your comments and questions are always valued and encouraged. If you have any other suggestions or techniques about PROC FORMAT that you would like to share, please feel free to drop me a line.

Jack N Shoemaker
ThotWave Technologies
Suite 202, 117 Edinburgh South
Cary, NC
Work Phone: 919 465 0322
Fax: 919 465 0323
Email: shoe@thotwave.com
Web: http://www.thotwave.com