Ten Things You Should Know About PROC FORMAT
Jack Shoemaker, Accordant Health Services, Greensboro, NC

ABSTRACT
The SAS® system shares many features with other programming languages and reporting packages. The programming logic found in the ubiquitous data step provides the mechanisms for assignment, iteration, and logical branching which rest at the core of any procedural language. Analytic data displays, like the humble frequency cross-tabulation produced by various PROC FREQ, may be replicated with varying degrees of success using any number of other products. PROC FORMAT is another matter. Somewhat like an enumerated data type; somewhat like a normalized and indexed reference table; it really has no exact analog in these other products and packages. There’s a lot you can do with PROC FORMAT. And, there’s a lot to know about PROC FORMAT. The aim of this paper is to provide insight on at least ten of those things which you should know.

1. IT’S JUST A SAS® CATALOG
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.

```sas
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 which 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-defined 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 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 which is used to read rather than display data.

2. USE THE DESCRIPTION
In version 8, the description attribute is left blank. In earlier versions, the description attribute contains some details about the format. 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 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:

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

The output will look something like this:

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

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 which you should find in the SAS® root directory which contains the SAS® executable file, sas.exe.

You can use PROC CATALOG to modify the description attribute of two members of the temporary catalog WORK.FORMATS.

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

In version 8, the description attribute is left blank. In earlier versions, the description attribute contains some details about the format. 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.
If your SAS® system administrators have acted in a responsible
duty, you will not be allowed to modify the common
LIBRARY.FORMATS catalog. So, the example above uses the
temporary format catalog called WORK.FORMATS which 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 contents display would look like this:

<table>
<thead>
<tr>
<th>#</th>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>AGE</td>
<td>FORMAT</td>
<td>Age Map</td>
</tr>
<tr>
<td>2</td>
<td>PHONE</td>
<td>FORMAT</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>AGE</td>
<td>FORMATC</td>
<td>Age Decoder</td>
</tr>
<tr>
<td>4</td>
<td>MYDATE</td>
<td>INFMT</td>
<td></td>
</tr>
</tbody>
</table>

3. EXAMINE THE CONTENTS

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.

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

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

<table>
<thead>
<tr>
<th>FORMAT NAME: AGE LENGTH:</th>
<th>MIN LENGTH: 1 MAX LENGTH: 40 D</th>
</tr>
</thead>
<tbody>
<tr>
<td>START</td>
<td>END</td>
</tr>
<tr>
<td>----------------+----------------+----</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>20</td>
</tr>
<tr>
<td>20&lt; 30</td>
<td>2</td>
</tr>
<tr>
<td>30&lt;HIGH</td>
<td>3</td>
</tr>
</tbody>
</table>

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 which 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.

4. UNLOAD THE CONTENTS

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.

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

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

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

5. THE REQUIRED COLUMNS

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
theses 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:

<table>
<thead>
<tr>
<th>Variable</th>
<th>Type</th>
<th>Label</th>
</tr>
</thead>
<tbody>
<tr>
<td>FMTNAME</td>
<td>Char</td>
<td>Format name</td>
</tr>
<tr>
<td>TYPE</td>
<td>Char</td>
<td>Type of format</td>
</tr>
<tr>
<td>START</td>
<td>Char</td>
<td>Starting value for format</td>
</tr>
<tr>
<td>END</td>
<td>Char</td>
<td>Ending value for format</td>
</tr>
<tr>
<td>LABEL</td>
<td>Char</td>
<td>Format value label</td>
</tr>
<tr>
<td>HLO</td>
<td>Char</td>
<td>Additional information</td>
</tr>
</tbody>
</table>

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

```sas
FMTNAME TYPE START END LABEL HLO
AGE N 0 20 1 HLO
AGE N 20 30 2
AGE N 30 HIGH 3 HLO
```

6. THE PUT() FUNCTION

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:

```sas
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.

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

You could replace code that looks like this:

```sas
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:

```sas
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:

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

or

```sas
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:

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

### 7. LOAD FORMAT FROM DATA SET OR TABLE

You may also create a user-defined format from an existing data set or data-base table. Imagine your vendor discount table have hundreds or thousands of entries. Manually coding this many entries 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 part 4 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:

```sas
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.

```sas
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,

```sas
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.
8. PICTURE CLAUSES
PROC FORMAT provides a special type of numeric format to place punctuation inside quasi-numeric data like phone numbers and social security numbers. It works by defining a mask into which the digits of a number are written. Picture clauses only work on numeric values. The following code fragment creates a user-defined picture format called PHONE which displays phone numbers with a set of parenthesis around the area code and a dash between the exchange and number.

```
proc format;
  picture phone
    low - high = '(999)999-9999'
    ( prefix = '(' );
```

Now consider the following set of phone numbers

```
data phones;
  infile cards; input phone;
cards;
  3363153714
  8009595605
  3153820
;run;
```

Using PROC PRINT to display these values using the PHONE. Picture format yields the following results.

```
PHONE
(336)315-3714
(800)959-5605
(000)315-3820
```

9. 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-formatted 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;
  value indate
    '00000000' = .Z
    '99999999' = .N
    other = [yymmd9.];
```

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

```
proc print data = sugme;
  format aDate otdate.;
run;
```

The results look like this:

```
aDate                       
 Some Zs                     
 Some 9s                     
 05JUN2000                   
```

10. MULTI-VALUE LABELS
The final 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 NW Molly
  XYZ SE Linda
```

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

```
proc format;
  value indate
    '00000000' = .Z
    '99999999' = .N
    other = [yymmd9.];
```

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

```
proc print data = sugme;
  format aDate otdate.;
run;
```

The results look like this:

```
aDate                       
 Some Zs                     
 Some 9s                     
 05JUN2000                   
```

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

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

```
proc format;
  value indate
    '00000000' = .Z
    '99999999' = .N
    other = [yymmd9.];
```

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

```
proc print data = sugme;
  format aDate otdate.;
run;
```

The results look like this:

```
aDate                       
 Some Zs                     
 Some 9s                     
 05JUN2000                   
```
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.

```sas
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,

```sas
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.

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

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

**CONCLUSION**

This paper has surveyed ten aspects of PROC FORMAT that should be in every SAS® programmers toolbox. It is not an exhaustive list of all that can be done with PROC FORMAT; nor are these even the ten most important things to know about PROC FORMAT. If you already use PROC FORMAT extensively, this paper may have provided you with one or two new ways to tackle a problem. If you haven’t begun to use PROC FORMAT yet in your day-to-day programming, this paper should provide some good examples on how to get started.