Paper TU02

PROC FORMAT is Our Friend

Erin Christen, AYW Training & Consulting, Knoxville, TN

ABSTRACT

For many programmers, formats are a magical wand that Data Management waves over the data to make everything 'pretty'. With the advent of electronic-submission-ready databases and CDISC standards even this application for PROC FORMAT has begun to fade. The average programmer uses PROC FORMAT only to make output 'pretty'. With the right approach PROC FORMAT can be so much more than just cosmetic. It can make many tasks simpler, make processes more automated, make changes quicker/easier to implement and ensure consistency among programs. This paper will cover the basics of PROC FORMAT, as well as some more advanced techniques. Best of all, the paper will provide numerous concrete examples of how PROC FORMAT can become a programmer's best friend.

INTRODUCTION

PROC FORMAT is a much more versatile tool than it would appear at first encounter. The only way to truly make full use of the tool is to have a thorough understanding of how it works and what options are available. To this end, we will first cover the most basic aspects of PROC FORMAT. Then we will touch on some more of the advanced options and techniques. Finally, several handy uses for PROC FORMAT will be provided.

THE BASICS

This section is intended for the beginner programmer (or for anybody who has shunned PROC FORMAT up to this point). We will cover the most basic aspects of working with formats.

LOCATING EXISTING FORMATS

Depending on the environment and set-up in which we are working, formats may already have been made available for our use. Before we can use PROC FORMAT most effectively, we should become familiar with the existing formats. To do this, we need to locate the format catalog.

By default, SAS looks for format catalogs in the WORK directory and in the LIBRARY directory. But, the options can also be adjusted so that SAS looks in a list of specified directories for format catalogs instead. So we must first determine what directories SAS searches for the format catalogs. Then we must determine whether or not format catalogs exist in those directories.

For the first step, we can use PROC OPTIONS to determine where SAS (in our environment) is looking for format catalogs. The code is as follows:

```
        proc options group=envfiles;
        run;
```

After running the code, in the LOG file, you will see a list of options and their current settings. Locate the FMTSEARCH option:

```
        FMTSEARCH=(WORK LIBRARY)
```

This will list the libraries in which SAS searches for format catalogs; whether or not format catalogs exist in these libraries remains to be seen in the second step.

For the second step, we need to physically check the libraries for a format catalog. Checking can be done in more than one way. One method would be to ascertain the physical path of the libraries listed in the FMTSEARCH. The following code (one of many methods) can be used to get the path:

```
        %put %sysfunc(pathname(library));
```

This code will print the full path associated with the libname LIBRARY to the LOG file. Then we can go to the directory and look for a file named FORMATS.sas7bcat (v8.1 and above).

An alternate method to check for format catalogs in the libraries listed in the first step would be the following code:

```
        proc contents data=library._all_
                      directory
                      memtype=catalog;
        run;
```

This will print to the LST file a list of all the catalogs located in the specified libname (in this example the libname LIBRARY). If there is catalog called FORMATS in the output list, then there is a formats catalog in that library.

VIEWING EXISTING FORMATS

Once we have located all existing format catalogs, we are most likely curious to see what formats are in these catalogs. The following code can be used to do that:

```
proc format library=<libname> fmtlib;
run;
*where <libname> = libname where a format catalog exists;
```

This code will print a map of the existing format catalog to the LST file. A small sample might look like this:

```
      FORMAT NAME: $SMOKE   LENGTH:   14   NUMBER OF VALUES:    3
   MIN LENGTH:   1  MAX LENGTH:  40  DEFAULT LENGTH  14  FUZZ:        0

START            END              LABEL  (VER. V7|V8   19DEC2005:22:31:50)

1                1                Current Smoker
2                2                Past Smoker
9                9                Never Smoked
```

This map provides an excellent overview of the formats already available for our use. Note if a format with the same name appears in multiple catalogs, the format from the first library listed in the FMTSEARCH option will be used.

SAS FORMATS

Regardless of the environment and set-up, there are also a whole host of formats provided with SAS. After digging into these, you'll find yourself amazed at what tools are already available. A complete list, along with descriptions and syntax, is available at Help, SAS Help and Documentation, Contents, SAS Products, Base SAS, SAS Language Dictionary, Dictionary of Language Elements, Formats (or Informats).

USING EXISTING FORMATS/INFORMATS

Using a format is fairly simple. There are two rules to remember: (1) if we are using a format we use the PUT() statement and using an informat we use the INPUT() statement, and (2) for character variables we must use a character format and for numeric variables we must use a numeric format.

Given a numeric variable called RACECD and a numeric format RACEF., to use the format to create a new variable the code would look like this:

```
newvar=put(racecd, racef.);
```

Similarly, for a character variable called SEXCD and a character format $SEXF., to use the format to create a new variable the code would look like this:

```
newvar=put(sexcd, $sexf.);
```

Similar code works for informats, replacing the PUT() with an INPUT().

We can also apply a format/informat directly to a variable by using the FORMAT/INFORMAT statement.

```
format racecd racef.;
```

or

```
format sexcd $sexf.;
```

The FORMAT/INFORMAT statement can be used in a DATA step and in most PROCS. Note that PROCs always use the formatted value of the variable for the analysis process.

CREATING A BASIC FORMAT (SIMPLE)

As noted, there are two types of formats, character and numeric. To create a basic numeric format, sample code looks like this:

```
proc format;
   value racef
          1='White'
          2='Black'
          3='Asian'
          9='Other'
          .='Unknown'
     ;
run;
*where racef is the name of the format to be created;
```

The values on the left hand side of the equal sign are the values in the data. The values on the right hand side are the labels we want to apply to the specified value. Note, the use of the . (dot) to indicate the label for the missing values. Without this, a missing value would have been formatted to a blank value, or in the case of creating a new variable with a PUT() statement, a missing value would appear as . (dot).

To create a basic character format, the code looks like this:

```
proc format;
     value $sexf
            'M'='Male'
            'F'='Female'
            ' '='Unknown'
      ;
run;
*where $sexf is the name of the format to be created;
```

Note the '$' on the format name which indicates a character format. Note also the use of the ' ' to indicate the label for the missing values. Without this, a missing value would have been formatted to a blank value.

In general, any word other than SAS reserved words can be used for the name of the format/informat. The name, however, cannot begin or end with a number. It can contain a number within the name, but it must have a character at the beginning and end. Also, By default in version 8 and later, the length of the format/informat name can be up to 32 characters. By resetting the VALIDFMTNAME option, the format/informat names can be forced to be 8 characters or less. To verify how long format/informat names can be, we can check the options with the following code:

```
proc options group=sasfiles;
run;
```

in the LOG file, the VALIDFMTNAME option indicates the current setting for format/informat names. LONG (the default) indicates that the allowed names may be up to 32 characters long:

```
VALIDFMTNAME=LONG
```

or FAIL indicates that names may be up to 8 characters long, and longer names will cause the program to fail:

```
VALIDFMTNAME=FAIL
```

or WARN indicates that names may be up to 8 characters long, and longer names will cause a WARNING to be issued, but the program will continue:

```
VALIDFMTNAME=WARN
```

CREATING A BASIC FORMAT (MULTIPLE VALUES)

Beyond just a simple one-to-one mapping as illustrated above, we can also map multiple values to a single label. The following code illustrates how this would work with a numeric format:

```
proc format;
     value phasecd
             1,2,3,4,5,6,7='Treatment'
             8,9,10,11,12='Follow-up'
      ;
 run;
```

Now the values 1,2,3,4,5,6,7 are all mapped to 'Treatment',and the values 8,9,10,11, and 12 are all mapped to 'Follow-up'. Any other values would display as they appear in the data.

We can apply a similar method for a character format:

```
proc format;
     value $phasecd
             'W1','W2','W3','W4','W5','W6','W7','W8'='Treatment'
             'W12','W16','W20','W24' = 'Follow-up'
      ;
 run;
```

Now values W1, W2, W3, W4, W5, W6, W7 and W8 are all mapped to 'Treatment', and values W12, W16, W20 and W24 are all mapped to 'Follow-up'.

CREATING A BASIC FORMAT (NUMERIC RANGES)

In addition, we can use ranges and lists of values to create formats. For example, we can apply a single 'label' to ranges of values with code like this:

```
      proc format;
         value abnlab
                 0 - < 9 ='OK'
                 9 – 12 ='WATCH'
                 12 < - <25 ='ALERT'
                 25 – 100 = 'EMERGENCY'
            ;
      run;
```
The number on the left hand of the hyphen is the bottom of the range. The number on the right hand side of the hyphen is the top of the range. Using just the hyphen indicates that the range is 'inclusive', that is the number on both ends of the range are mapped to the value. Using the (<) sign, excludes either end of the range. (Note that the sign (<) is always the same but its location in reference to the hyphen makes it a greater than or less than.) In the example above, values of 0 to less than but not including 9 are mapped to 'OK'. Values of 9 to 12, including 9 and 12 are mapped to 'WATCH'. Values of greater than and not including 12 to less than and not including 25 are mapped to 'ALERT'. Values of 25 to 100, including 25 and 100 are mapped to 'EMERGENCY'.

CREATING A BASIC FORMAT (NUMERIC RANGES WITH HIGH AND LOW)

For the range above, to account for all values, we would need to know how small our values might get and how large our values might get. To make a more flexible format, we can use the keywords HIGH and LOW. Using the format above, we could account for all possible values without knowing precisely how big or small they might get by making the following additions:
```
      proc format;
         value abnlab
                 low - < 0 = 'TOO LOW – RECHECK'
                 0 - < 9 ='OK'
                 9 – 12 ='WATCH'
                 12 < - <25 ='ALERT'
                 25 – 100 = 'EMERGENCY'
                 100 < - high = 'TOO HIGH – RECHECK'
            ;
      run;
```
CREATING A BASIC FORMAT (KEYWORD OTHER)

We can also use the keyword OTHER to account for all values. An alternate modification to the first format might be the following:
```
      proc format;
         value abnlab
                 0 - < 9         = 'OK'
                 9 – 12          = 'WATCH'
                 12 < - <25      = 'ALERT'
                 25 – 100        = 'EMERGENCY'
                Other            = 'ERROR – RECHECK'
            ;
      run;
```
Now, all the values which are not accounted for in the ranges will be displayed as 'ERROR – RECHECK'. Note that missing values are included in OTHER and will be displayed as 'ERROR – RECHECK.' If this is not what we desire, then we need to explicitly account for missing values in our format, like this:
```
      proc format;
         value abnlab
                 0 - < 9         = 'OK'
                 9 – 12          = 'WATCH'
                 12 < - <25      = 'ALERT'
                 25 – 100        = 'EMERGENCY'
                 .               = 'BLANK'
                Other            = 'ERROR – RECHECK'
            ;
      run;
```

CREATING A BASIC FORMAT (CHARACTER RANGES)

We can apply the same basic principles of ranges to character formats as well. We could create the following format:

```
        proc format;
           value $abc
                'A' - <'K'   = 'First 10'
                'K' - <'U'   = 'Second 10'
                'U' - high   = 'Remainder'
                 Other       = 'Error - Recheck'
             ;
        run;
```

Any values which come after A (including A) and before K (not including K), in an alphabetical sort, will be labeled as 'First 10'. Any values which come after K (including K) and before U (not including U), in an alphabetical sort, will be labeled as 'Second 10'. Any values which come after 'U' (including U) will be labeled as 'Remainder'. All other values (including missing) will be displayed as 'Error – Recheck'. Keep in mind that in an alphabetical sort, SAS sorts numeric values before A. So, in this example, missing values and any values beginning with numbers will be labeled as 'Error – Recheck'.

The remainder of the rules for ranges and keywords detailed above for numeric values apply in a similar manner to character values.

CREATING A BASIC INFORMAT

Creating informats is similar to creating formats except the VALUE statement is replaced with an INVALUE statement. One important distinction is that the right-hand side of the equals sign in a VALUE statement (the 'label' side) for a format must be character. It cannot be numeric. In creating an informat, the right-hand side of the equals sign in the INVALUE statement can be numeric. (See Handy Use #9 below.)

A second important distinction is that for informats the type of informat that we use/create (character vs numeric) indicates the type of the OUTPUT variable. This is the opposite of formats. For formats, the type we use/create indicates the type of the INPUT variable. Using one of our format examples from above:

```
        proc format;
           value racef
                     1='White'
                     2='Black'
                     3='Asian'
                     9='Other'
                     .='Unknown'
              ;
        run;
```

Here we are converting a numeric input variable with values of 1,2,3 or 9 to a character label of 'White','Black','Asian' or 'Other'. Based on the numeric type of the input variable, we create a numeric format RACEF.

If we wanted to use an informat to make the same conversion, we would instead create a character informat based on the type (character) of the label.

```
        proc format;
           invalue $racef
                     1='White'
                     2='Black'
                     3='Asian'
                     9='Other'
                     .='Unknown'
              ;
        run;
```

As a side note, in general, it is best to use a format when converting from numeric to character. Informats are better suited to converting character to numeric or character to character. Using an informat to convert from numeric to character and/or numeric to numeric will generate a "Note: Numeric values have been converted to character values" message in the program LOG.

STORING FORMATS

In all the examples above, the PROC FORMAT statement creates the format and stores it in a format catalog in the WORK directory. As such, it is available throughout the current SAS session. After leaving the session, the format(s) will no longer be available. If we want to permanently store the format for use in later SAS sessions (aka other programs), we simply add a LIBRARY option to the PROC FORMAT statement.

In our above examples, we can replace the line of code:
```
        proc format;
```
with the line of code:
```
        proc format library=<libname>; *where <libname> is the libname in which to store
                                         the format;
```
With this addition, instead of storing the formats in a catalog in the WORK directory, SAS will store the formats in a catalog in the specified directory (<libname>).

ADVANCED TECHNIQUES

Now that we've mastered the basics, we can move on to the more advanced aspects of PROC FORMAT.

USING THE _ERROR_ OPTION

With the _ERROR_ option, whenever the specified value(s) is encountered and ERROR message will be issued in the LOG file. The _ERROR_ option only applies with informats (not formats.) Example code:
```
        proc format;
           invalue racef
                   '1'='White'
                   '2'='Black'
                   '3'='Asian'
                   '4','5','6','7','8'=_ERROR_
                   '9'='Other'
                   ' '='Unknown'
             ;
        run;
```
When we use the informat, if it encounters a value of 4,5,6,7 or 8 an ERROR message will be printed to the log.
```
        NOTE: Invalid argument to function INPUT at line 107 column 8.
        var=5 var2=  _ERROR_=1 _N_=2
```
(See Handy Use #8 below.)

USING THE _SAME_ OPTION

By default, if a value in the data is not included in the informat (or format), when we apply the informat (r format) the omitted value will be displayed exactly as it appears in the data. The _SAME_ option is an alternate method for stating this default. Like the _ERROR_ option above, it is only applicable to INFORMATS. This code:
```
        proc format;
             invalue phasecd
                    1,2,3,4,5,6,7='Treatment'
                    8,9,10,11,12='Follow-up'
              ;
        run;
```
Generates the same results as this code:
```
        proc format;
             invalue phasecd
                    1,2,3,4,5,6,7='Treatment'
                    8,9,10,11,12='Follow-up'
                    Other=_SAME_
              ;
        run;
```

Where the _SAME_ option proves necessary is if we want to use the _ERROR_ option. If we use the first code (with any omitted values) with the _ERROR_ option, like this:
```
        proc format;
             invalue phasecd
                    '1','2','3','4','5','6','7'='Treatment'
                    '8','9','10','11','12'='Follow-up'
                    Other=_ERROR_
              ;
        run;
```

When the omitted values are encountered an error will be issued. If we instead include the omitted values with the _SAME_ option:

```
       proc format;
            invalue phasecd
                   '-1', '0' = _SAME_
                   '1','2','3','4','5','6','7'='Treatment'
                   '8','9','10','11','12'='Follow-up'
                   other=_ERROR_
             ;
        run;
```

The -1 and 0 values will now appear exactly as they are in the data, rather than issuing an error to the LOG. All other values will still issue an error in the LOG. (See #8 below.)

USING THE DEFAULT OPTION

Occasionally, there will appear a NOTE in the LOG about "the length of the format is defaulted to XX". This happens particularly when the length of the character strings on the output (right) hand side are long or when we use an existing format to create a new format (see below). The warning can be removed by specifying with the DEFAULT option how long we want the format to be. It is the same general idea as when we apply a length to a variable. The DEFAULT option applies to the format display (eg. the information on the right hand side). The code looks like this:

```
       proc format;
            value phasecd (default=15)
                   -1, 0 = _SAME_
                   1,2,3,4,5,6,7='Treatment'
                   8,9,10,11,12='Follow-up'
                   other=_ERROR_
             ;
        run;
```

This code sets the length of the formatted values to 15 characters. It is not necessary to include the DEFAULT option every time. Adding the DEFAULT can remove the warning. It can also help reduce dead spaces appearing at the end of output. Note that if the length of the label (value on the right-hand side) is longer than the length set by the DEFAULT option the label will be truncated.

CREATING A PICTURE FORMAT (NUMBER DISPLAY)

Using the PICTURE statement in PROC FORMAT allows us to leverage the power of PROC FORMAT. We can make labels look exactly like we want them to look. A picture format works the same as a basic format on the left hand side (eg. the specification of ranges and values), the difference falls on the right hand side (eg. the label display).

For the label display there are three things to know, 1) using a '9' (or any non-zero number) as a digit selector will force the number to appear (i.e. will include leading zeros), 2) using a 0 as a digit selector will only display the number if that digit is in the number (i.e. no leading zeros), and 3) using a character string, will display that character string exactly as included in the format.

The best way to understand a picture format is by looking at examples:
A format like this one:

```
       proc format;
            picture testA
                   Low-high = '999.99'
            ;
        run;
```

Yields these results when format TESTA is applied to VAR when creating NEWVAR:

```
          VAR           NEWVAR
        1234.567         234.56
         123.456         123.45
         123.450         123.45
         123.400         123.40
         123.000         123.00
         12.345          012.34
         12.340          012.34
         12.300          012.30
         12.000          012.00
         1.234           001.23
         1.230           001.23
         1.200           001.20
         1.000           001.00
```

Note the leading zeros when the value before the decimal is less than 3-digits long. Also note in the first row, when the number before the decimal is more than 3-digits long, the extra digits are truncated. No message appears in the long alerting to this occurrence. Also note that the values are truncated, rather than rounded, when the number of decimals exceeds the number of places indicated in the format. (Refer to ROUND option below to change this.)

A format like this one:
```
       proc format;
            picture testB
                   Low-high = '000.00'
            ;
        run;
```
Yields these results when format TESTB is applied to VAR when creating NEWVAR:
```
           VAR           NEWVAR
        1234.567         234.56
         123.456         123.45
          123.450        123.45
          123.400        123.40
          123.000        123.00
          12.345         12.34
          12.340         12.34
          12.300         12.30
          12.000         12.00
          1.234          1.23
          1.230          1.23
          1.200          1.20
          1.000          1.00
```
Note that the difference between this and the previous example is that the leading zeros no longer appear when the value is less than 3-digits long. However, the number of decimal places always remains at two whether I use a 9 or a 0.

A format like this one:
```
       proc format;
            picture testC
                   Low-high = '099.00'
            ;
        run;
```
Yields these results when format TESTC is applied to VAR when creating NEWVAR:
```
           VAR           NEWVAR
        1234.567         234.56
         123.456         123.45
          123.450        123.45
          123.400        123.40
          123.000        123.00
          12.345         12.34
          12.340         12.34
          12.300         12.30
          12.000         12.00
          1.234          01.23
          1.230          01.23
          1.200          01.20
          1.000          01.00
```
Note that up to 3-digits may appear before the decimal, but only two are zero-filled – as indicated by the 9's in the format definition.

CREATING A PICTURE FORMAT (APPENDING TEXT)

A text string can be added AFTER (but not BEFORE) the numbers by simply including the text in the format. A format like this:
```
       proc format;
            picture testD
                   Low-high = '099.00 - test'
            ;
        run;
```
Yields these results when format TESTD is applied to VAR when creating NEWVAR:
```
           VAR           NEWVAR
        1234.567         234.56 - test
         123.456         123.45 - test
         123.450         123.45 - test
         123.400         123.40 - test
         123.000         123.00 - test
         12.345          12.34 - test
         12.340          12.34 - test
         12.300          12.30 - test
         12.000          12.00 - test
         1.234           01.23 - test
         1.230           01.23 - test
         1.200           01.20 - test
         1.000           01.00 - test
```
Note that the number is formatted exactly the same as in the previous example (TESTC) and the string is appended. As noted earlier, text can only be appended AFTER the number in this way. To append text BEFORE the number we must use the PREFIX option detailed later.

CREATING A PICTURE FORMAT (CHARACTER DISPLAY)

A character string can be displayed for certain values, while displaying numbers for the other values. A format like this:
```
      proc format;
           picture testE
                  Low-<1000 = '099.00'
                  1000 - high = 'Value too high'
           ;
       run;
```
Yields these results when format TESTE is applied to VAR when creating NEWVAR:
```
          VAR            NEWVAR
        1234.567         Value too high
         123.456         123.45
         123.450         123.45
         123.400         123.40
         123.000         123.00
         12.345          12.34
         12.340          12.34
         12.300          12.30
         12.000          12.00
         1.234           01.23
         1.230           01.23
         1.200           01.20
         1.000           01.00
```
Note that the text string appears exactly as it appears in the format. This is true as long as there are no numbers in the string. If there are numbers in the string, they will be formatted as was done in format TESTD above and any characters before the numbers will be ignored. As a side note, format TESTE provides an alert for the problem noted in format TESTA when the 4 digit value was truncated.

CREATING A PICTURE FORMAT (ROUND OPTION)

By default, when a picture format is applied, it truncates the data. (As a side note, by default when an existing SAS format is applied like 5.1 or 7.3, the data is rounded.) In general, rounding is preferred to truncating. To make a picture format round instead of truncating, we simply add the ROUND option:

A format like this one:
```
      proc format;
           picture testF (round)
                  Low-high = '000.00'
           ;
       run;
```
Yields these results when format TESTF is applied to VAR when creating NEWVAR:
```
          VAR            NEWVAR
        1234.567         234.57
         123.456         123.46
         123.450         123.45
         123.400         123.40
         123.000         123.00
         12.345          12.35
         12.340          12.34
         12.300          12.30
         12.000          12.00
         1.234           1.23
         1.230           1.23
```

```
 1.200           1.20
 1.000           1.00
```

Note that the final decimal digit is now rounded, rather than truncated. (Compare these results to those from TESTB above, without the round.) Also note that in the first row, the value with 4 digits before the decimal is still truncated on the front and no message will appear in the log.

CREATING A PICTURE FORMAT (PREFIX OPTION)

With a picture format, character strings can be displayed after the digits as is shown above in format TESTD. Character strings cannot be displayed before the digits. A format like this:

```
proc format;
     picture testG
            Low-<1000 = '099.00'
            1000 - high = 'Value of 0000.00 too high'
     ;
run;
```

Yields these results when format TESTG is applied to VAR when creating NEWVAR:

```
   VAR           NEWVAR
1234.567         1234.56 too high
 123.456         123.45
 123.450         123.45
 123.400         123.40
 123.000         123.00
 12.345          12.34
 12.340          12.34
 12.300          12.30
 12.000          12.00
 1.234           01.23
 1.230           01.23
 1.200           01.20
 1.000           01.00
```

Notice that the "Value of " portion of the string is removed and the number appears formatted as indicated by the "0000.00" with the string "too high" appended to the end.

The PREFIX option allows us to put a character string before the numbers. Replacing the code above with the following will fix the error and display the output as we intended:

```
proc format (default=30);
     picture testH
            Low-<1000 = '099.00'
            1000 - high = '0000.00 too high' (prefix='Value of ')
     ;
run;
```

Yields these results when format TESTH is applied to VAR when creating NEWVAR:

```
   VAR           NEWVAR
1234.567         Value of 1234.56 too high
 123.456         123.45
 123.450         123.45
 123.400         123.40
 123.000         123.00
 12.345          12.34
 12.340          12.34
 12.300          12.30
 12.000          12.00
 1.234           01.23
 1.230           01.23
 1.200           01.20
 1.000           01.00
```

Note that the DEFAULT option is applied in the format to avoid truncation. This may or may not be necessary, but better safe than sorry.

CREATING A PICTURE FORMAT (NOEDIT OPTION)

The NOEDIT option can be used to help avoid some collisions with the data in the PICTURE format. In our example TESTH have, suppose instead of "Value of 0000.00 is too high", we want to display the text "Value >1000 is too high." Instinct and our lesson about the PREFIX option above would indicate that the format should look like this:

```
        proc format;
             picture testI
                    low-<1000 = '099.00'
                    1000 - high = '1000 too high' (prefix='Value of >')
             ;
         run;
```
But, SAS thinks we want to have the digit selectors 1000 followed by a "too high" and the result would look like this:
```
           VAR            NEWVAR
         1234.567         Value of >1234 too high
```

To force SAS to recognize this as a single text string, we add the NOEDIT option.
```
        proc format;
             picture testJ
                    Low-<1000 = '099.00'
                    1000 - high = 'Value of >1000 too high' (noedit)
             ;
         run;
```
Now when SAS encounters values of 1000 or more, the entire string "Value of >1000 too high" will be displayed without any modifications. The results look like this:
```
           VAR            NEWVAR
         1234.567         Value of >1000 too high
```

CREATING A PICTURE FORMAT (MORE OPTIONS)

There are several other options that may be helpful with picture formats, including MAX, MIN, FUZZ, FILL, etc. Due to space, only the more commonly used options are included here. Complete details for these additional options can be found by going to Help, SAS Help and Documentation, Contents, SAS Products, Base SAS, SAS Procedures, Procedures, The FORMAT Procedure, PICTURE Statement.

CREATING A FORMAT FROM AN EXISTING FORMAT

In addition to creating formats of our own, we can also use existing formats (both SAS formats and/or user-created formats) to create new formats.

Suppose our study format catalog has a format for treatment group called TRTGRP which labels values like this:
0=Control
1=Drug
2=No Treatment
And we want a 4th value with a label of Total. We could create a new format like this:
```
         proc format;
            value trtgrpB
              0='Control'
              1='Drug'
              2='No Treatment'
              3='Total'
            ;
        run;
```
Or we could use the existing format to create the new format, like this:
```
         proc format;
               value trtgrpB
                     3='Total'
                     Other=[trtgrp.]
               ;
        run;
```
Now for the value of 3, 'Total' will be displayed and for all other values, the existing TRTGRP format will be used.

We can also re-assign values. Suppose in the example above we want to add a footnote to the 'No Treatment' label. I could again create a whole new format. Or use the existing one, like this:
```
         proc format;
               value trtgrpC
                     2='No Treatment (a)'
                     Other=[trtgrp.]
               ;
        run;
```
Now, for the value of 2, this new display will be used, but for all other values the existing TRTGRP will be used. We could, of course, do both of these at once with:

```
        proc format;
               value trtgrpD
                      2='No Treatment (a)'
                      3='Total'
                      Other=[trtgrp.]
               ;
        run;
```
Creating formats from existing formats in this way can help ensure consistency and allow for simple changes across outputs.

USING CNTLOUT TO MAKE A DATASET FROM FORMATS

We can also output the values of formats into a dataset. This allows for more complicated manipulations with the values or to use the information for other purposes (see Handy Use #3 below). The basic code to output all the formats into a dataset looks like this:
```
        proc format library=work cntlout=fmtds;
        run;
```
This creates a dataset called FMTDS which contains all the format information from the format catalog stored in the WORK directory. We can look at format catalogs in other directories by changing the LIBRARY= option. We can also name the output dataset with any acceptable dataset name.

The output dataset created from the format catalog contains the name of all the formats in the catalog (variable:FMTNAME) with one record for each of the labels in the format. The variable LABEL contains the information from the right-hand side of the equal signs. The variables START and END contain the top and bottom, respectively, of the range to which the specified label applies. There are also other variables PREFIX, NOEDIT, etc. which indicate the additional options that are applied to the format. Once the dataset is generated, it can be manipulated just like any other SAS dataset.

The SELECT and EXCLUDE options can be used to specify which formats to include in the output dataset. The SELECT option specifies which formats to include; conversely the EXCLUDE option specifies which formats to exclude. Only one can be used at a time.

This code:
```
        proc format library=work cntlout=fmtds;
            select racecd testA;
        sun;
```
Creates the same final dataset as this code:
```
        proc format library=work cntlout=fmtds;
        run;
        data fmtds;
            set fmtds(where=(upcase(fmtname) in ('RACECD','TESTA')));
        run;
```
This code:
```
        proc format library=work cntlout=fmtds;
            exclude racecd testA;
        run;
```
Creates the same final dataset as this code:
```
        proc format library=work cntlout=fmtds;
        run;
        data fmtds;
            Set fmtds(where=(upcase(fmtname) not in ('RACECD','TESTA')));
        run;
```
Using the SELECT or EXCLUDE options can substantially decrease the size of the output dataset (and processing time) when we are dealing with large, centralized format catalogs.

USING CNTLIN TO MAKE FORMATS FROM A DATASET

We can reverse the process and create formats from a dataset. The code looks like this:
```
        proc format library=work cntlin=mkfmtds;
        run;
```
Which takes the dataset called MKFMTDS and adds the detailed formats to the format catalog in the work directory. We can put the formats in a format catalog in another directory by changing the LIBRARY= option.

We can use any acceptable name for the input dataset but the dataset must contain specific variables. These are the same variables that are included in the dataset created by the CNTLOUT option (eg FMTDS above). At a minimum the input dataset (eg. MKFMTDS) must contain the FMTNAME, START, END and LABEL variables. The other variables, PREFIX, NOEDIT, LENGTH, etc. only need to be added if those are options we want to apply to the format.

For example, to create this format
```
      proc format;
         value racef
                 1='White'
                 2='Black'
                 3='Asian'
                 9='Other'
                 .='Unknown'
            ;
      run;
```
through a dataset, the dataset would look like this:
```
      FMTNAME       START       END      LABEL
      RACEF           1          1        White
      RACEF           2          2        Black
      RACEF           3          3        Asian
      RACEF           9          9        Other
      RACEF           .          .        Unknown
```
The important points to note are: 1) the dataset for this particular format needs only to contain the basic 4 variables, 2) One observation is created for each of the labels we want to apply, and 3) the value of FMTNAME is the same for all the applicable labels. We can create multiple formats from a single input dataset, using the FMTNAME variable to control which values are applied to which format.

Note also that since each label applies only to a single value in this particular format, the START and END values have been set to the same value. START and END must both have an applicable value. If we used START=1 and END=. for the LABEL=White, SAS would have tried to create a format which applied the label "White" to a range from 1 to missing. Attempting this would have made SAS error out, because the end of the range is smaller than the beginning of the range.

Similarly, we can create some of our other previous example formats via a dataset.
Example 1: For this format:
```
      proc format;
         value trtgrpC
                   2='No Treatment (a)'
                   Other=[trtgrp.]
             ;
      run;
```
We use a dataset that looks like this:
```
      FMTNAME       START       END      LABEL                      HLO
      TRTGRPC         2          2        No Treatment (a)
      TRTGRPC         .          .        TRTGRP.                    OF
```
Note the additional variable HLO which is set to OF for the "Other" label. The "O" part of the value in this variable indicates the START and END should be OTHER. The "F" part of the value in this variable indicates that the LABEL is an existing format rather than a character string. The values can be used individually to apply the separate criteria.

Example 2: For this format:
```
      proc format;
            picture pctf
                   Low-high = '000.0%)' (prefix='(')
             ;
       run;
```
We use a dataset that looks like this:
```
      FMTNAME       START       END      LABEL                      HLO   PREFIX   TYPE
        PCTF          .          .       000.0%)                    LH      (        P
```
The "L" in the HLO variable indicates that the START should be LOW. Similarly the "H" in the HLO variable indicates that the END should be HIGH. The PREFIX variable stores the value of the PREFIX option. The TYPE variable set to P indicates that this is a PICTURE format (as opposed to a numeric format (N) or a character format (C)). In our previous example, the TYPE is set to N (numeric) based on the values in START and END.

Example 3: For this format:

```
proc format;
   value abnlab
           0 - < 9 ='OK'
           9 - 12 ='WATCH'
           12 < - <25 ='ALERT'
           25 – 100 = 'EMERGENCY'
          Other = 'ERROR – RECHECK'
     ;
run;
```

We use a dataset that looks like this:

```
FMTNAME        START   END      LABEL                 HLO   SEXCL  EEXCL
ABNLAB           0       9       OK                          N      Y
ABNLAB           9      12       WATCH                       N      N
ABNLAB          12      25       ALERT                       Y      Y
ABNLAB          25     100       EMERGENCY                   N      N
ABNLAB           .       .       ERROR – RECHECK        O    N      N
```

As before there is one observation for each label. Also as before, for the last label the HLO='O' indicates that the specified label should be applied to the "OTHER" values. The new variables here are the SEXCL (Start Exclusion) and EEXCL (End Exclusion). These variables indicate whether or not the value specified in the START (SEXCL) and END (EEXCL) variables should be included (N) or excluded (Y) from the range. Note that wherever the (<) sign appears in the original PROC FORMAT, the corresponding value of SEXCL/EEXCL is set to Y in the dataset.

An excellent way to further explore using a dataset and the CNTLIN option to create formats is to create formats in the standard way with PROC FORMAT and the VALUE, INVALUE or PICTURE statement. Then use the CNTLOUT option to create a dataset from these formats. By examining the values of the variables in the output dataset created this way, we can quickly become familiar with the necessary values and variables we need in our dataset to use with the CNTLIN option. (See Handy Use #4.)

HANDY USES

As promised, now that we've covered the basic (and not so basic) aspects of PROC FORMAT, we come to everybody's favorite part - how to put this new found knowledge to use to make our work day lives easier. Following are 9 handy uses for PROC FORMAT.

HANDY USE #1: QUICK PROC FREQ

In doing exploratory analyses, the need often arises to collapse the data into categories. Given the nature of exploratory analyses, the categories may keep changing. Suppose, for example, we are trying to figure out how best to stratify our population by age. Basic code to look at the number of subjects that fall into a given age category would usually look like this:

```
data example;
      set raw.source;
      length agecat $20;
       if . < age < 5 then agecat='0 to <5';
       else if 5 <= age < 10 then agecat='5 to <10';
       else if 10 <= age < 20 then agecat='10 to <20';
       else if 20 <= age < 30 then agecat='20 to <30';
       else if 30 <= age < 50 then agecat='30 to <50';
       else if age >=50 then agecate='50 and above';
run;
proc freq data=example;
     table agecat;
run;
```

Using PROC FORMAT we could shorten the code required:

```
proc format;
     value agecat
           0 - < 5 = '0 to <5'
           5 - < 10 = '5 to <10'
          10 - < 20 = '10 to <20'
          20 - < 30 = '20 to <30'
          30 - < 50 = '30 to <50'
          50 – high = '50 and above'
     ;
run;
```

```
        proc freq data=raw.source;
             format age agecat.;
              table age;
         run;
```
By default PROC FREQ (and all other PROCs) uses the formatted value of a variable for classification (aka grouping). So the counts that appear in this PROC FREQ are identical to the counts that appear in the previous PROC FREQ. The code using PROC FORMAT gives us several advantages: 1) saved key strokes, 2) the ability to run our PROCS directly off the raw.source dataset, without the intermediary data step (saving processing time if the dataset is very large), and 3) unaccounted for values don't get lost in the "MISSING" category of the PROC FREQ, they appear in the PROC FREQ output as the actual value.

As a side note, if we want PROC FREQ (or other PROCs) to use the unformatted value of a variable that has a format already assigned (suppose we had a variable RACECD with the format RACEF associated with it), we can temporarily 'remove' the format with code like the following:
```
        proc freq data=raw.demo;
             format racecd;
              table racecd;
         run;
```
Now PROC FREQ will use the actual values of the RACECD rather than the formatted values. But the format RACEF will still be associated with the variable RACECD.

HANDY USE #2: DISPLAYING PERCENTAGES

Displaying percentages complete with parenthesis and the percent sign is a common task in clinical programming. The standard data step code may look like this:
```
        disppct='('||put(pct, 5.1)||'%)';
```
We can turn this into a more automatic process by using a PICTURE format. The format looks like this:
```
        proc format;
              picture pctf (round)
                      low-high = '000.0%)' (prefix='(')
              ;
         run;
```
Then in our data step we can use this code to generate the same results we had before:
```
        disppct=put(pct, pctf.) ;
```
If we add this format to our universal format catalog and use it in all our programs to display percentages, we can easily ensure that all the percentages are uniformly displayed. Also, if we decide at any point to remove the parenthesis or remove the percent sign we can make the change in all outputs by making a simple change to the format catalog.

This PICTURE format also points out a couple of things that need to be noted about using the PREFIX option. First, the PREFIX option puts the character strings flush with the digits. It does not fill with space. In our example above, if we have (1.1%) and (100.0%), they will display exactly like that, rather than as (1.1%) and (100.0%). This may cause some headaches with lining things up in the output.

Second, we must include a digit selector (0,9) in the number sufficient to cover the number of characters in the PREFIX or specify the DEFAULT option with a sufficient length. If we do not and the number takes up all the digits, the prefix may not be included. For example, the PCTF format above omits the opening parenthesis when 100.0 is displayed. We can correct this either by adding a digit selector to account for the prefix:
```
        picture pctf (round)
               low-high = '0000.0%)' (prefix='(')
        ;
```
Or by specifying a long enough length in the DEFAULT option:
```
        picture pctf (round default=8)
               low-high = '000.0%)' (prefix='(')
        ;
```

HANDY USE #3: ZERO-FILLING (AKA 'DUMMYING UP') OUTPUT DISPLAY

In programming tables, it is often a requirement that even if no subjects fall into a given category, we still want to display that category on the output with zero values. There are a variety of ways to add those rows into the data, also referred to as 'dummying up' the data. PROC FORMAT can provide both a quick and easy method to dummy up the data. (This idea originated in PharmaSUG 2005 Coder's Corner paper #22 by Stacey Phillips and Gary Klein. Please refer to this paper for a more complete description. The idea is presented in brief here.)

For this example, we will look at creating zero rows for the display of the counts for the variable RACED. The format associated with this variable is RACEF. The first step of the dummying up process is to create a dataset with the information about the associated format, RACEF. To do this we use the CNTLOUT option of PROC FORMAT. (Refer to USING CNTLOUT TO MAKE A DATASET FROM FORMATS for complete details.) Assuming our formats are kept in a library named FMTLIB, the code would look like this:

```
        proc format library=fmtlib cntlout=dummy;
            select racecf;
        run;
```

Now that we have the dataset DUMMY, we can use the following code to keep the values and the labels associated with those values:

```
        data dummy;
            set dummy;
            keep start label;
            rename start=racecd;
        run;
```

In two simple steps we have created our DUMMY dataset, which we can now merge with our original dataset containing the counts to create an observation for each possible value of RACECD.

In addition to being quick and easy, using PROC FORMAT automates the process. Whenever the related format (here RACEF) is updated in the format catalog, the 'dummying up' process in the table is automatically updated to reflect the additional (or removed) values.

HANDY USE #4: CREATING FORMATS FROM THE DATA

In a CDISC world where data management no longer sends us format catalogs but instead sends us code/decode variables in the data, we may still need formats for certain outputs. Instead of hand-coding all the necessary formats, we can create the formats directly from the code/decode variables in the data.

Given, for example, dataset DEMOG with a variable RACECD which has the coded value of the race (1,2,3) along with the variable RACE which has the decoded value of the race (White, Black, Asian), I can quickly create a format called RACEF using the CNTLIN option. (Refer to USING CNTLIN TO MAKE FORMATS FROM A DATASET for complete details.)

The first step is to prepare the dataset that the CNTLIN option needs:

```
        *get one record for each possible code/decode combination;
       proc sort data=demog out=mkfmt nodupkey;
           by racecd race;
        run;
       *prepare the dataset for CNTLIN;
        data mkfmt;
             set mkfmt;
             fmtname='RACEF';
             start=racecd;
             end=racecd;
             rename race=label;  *use rename to avoid truncation;
        run;
```

Once the dataset is prepared, we simply read it in to a format:

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

And now we have a format called RACEF which we can use in other programs to format the RACECD variable when we need it.

Keep in mind, however, that this only generates a format based on the values in the existing data. The code should be written/maintained so that the format is re-generated anytime the data is updated to ensure that all values are included. Towards this end, it may be useful to create a dataset (from the CNTLOUT option on PROC FORMAT) which stores all the current information for the formats created. This dataset can be used as a base to which new values are appended without losing any existing data. This way, using our example from above, if we have one study with RACECD=1 and 3 only. And then another study with RACECD=1 and 2 only. We can combine this information in a cumulative manner and end up with a universal format that accounts for RACECD=1, 2 and 3, so that our universal format catalog improves with age.

HANDY USE #5: CREATING CATEGORICAL VARIABLES (REPLACING IF, THEN STATEMENTS)

Creating categorical variables can also be simplified with PROC FORMAT. If the variable only needs to be created once, then the standard IF, THENs are just as simple. If, however, the categorical variable needs to be created in multiple places within the same study, PROC FORMAT can provide an alternate method.

An excellent example would be the process of creating visit windows based on the study day. Since visit windows usually need to be created in all datasets, the process is typically tucked into a macro which is then called in the individual programs. A simple example for generating visit windows might look like this:

```
        %macro mkwndw(day=, wnd=);
             if . < &day <= 10 then &wnd=1;
             else if 10 < &day <= 21 then &wnd=2;
             else if 21 < &day <= 42 then &wnd=3;
             else if &day > 42 then &wnd=4;
        run;
        %mend mkwndw;
```

Then, in the individual programs, the macro would be called as:

```
        data inds2;
             set inds;
             %mkwndw(day=studyday, wnd=wkwndw);
        run;
```

If macros are frowned upon at our company or we just prefer to keep things as self-contained as possible, PROC FORMAT provides another solution. Instead of the code above, we can build a format like this:

```
        proc format;
            value window
                 0 – 10      = '1'
                 20 < - 21   = '2'
                 21 < - 42   = '3'
                 42 < - high = '4'
            ;
        run;
```

Then, in the individual programs, the format can be used to create the same results as the macro above:

```
        data inds2;
            set inds;
            wkwndw=input(put(studyday, window.), best.);
        run;
```

If, as is usually the case, the process of creating the visit window or other categorical variable varies slightly from dataset to dataset, we can create multiple formats and use the appropriate one in each applicable program.

This idea can be expanded to replace IF, THEN statements in a variety of situations.

HANDY USE #6: CREATING CATEGORICAL VARIABLES (CONSISTENCY AND EASE OF CHANGE)

Using PROC FORMAT to create standard categorical variables (like age categories or collapsed races) which are the same in all studies, across all drugs, can make the creation of the variables consistent and much easier to change.

Consider the AGECAT variable (or some variation on that name). This is the variable that indicates AGE < 65 and AGE >=65. This variable must be generated in almost every study (at least all adult studies.) So at some place in every study the following code should appear:

```
        if . < age < 65 then agecat=1;
        else if age >=65 then agecat=2;
```

Programmers, being individuals, sometimes allow variations to creep in:

```
        if age < 65 then agecat=1;      *now missing ages are being include in 1;
        else if age >=65 then agecat=2;
```

Or

```
        if . < age <= 65 then agecat=1;      *now we've put 65 in 1, instead of 2;
        else if age > 65 then agecat=2;
```

We can prevent all these problems in all studies by creating this variable with a format instead:

```
        proc format;
             value agecat
                 0 - < 65 = '1'
                 65 – high = '2'
             ;
        run;
```

Then in the programs use this code:

```
        agecat=input(put(age, agecat.), best.);
```

We get the exact same results every time without the potential for mistakes. If the age categories of interest should ever change (not so many years ago they were <60 and >=60), we can make a simple adjustment in a single place in the format catalog and the change will be implemented in all programs.

The same idea of consistency and ease of change exists in a host of other variables that appear in many studies and/or many datasets. PROC FORMAT can give us that consistency and ease of change.

HANDY USE #7: CHECKING FOR MISSING RECORDS

In the process of cleaning and validating data, one important step is to ensure that variables are completed. A quick check for the completeness of variables is a count of the missing vs non-missing values. PROC FORMAT can reduce the process of a missing check on all the variables in a dataset to a few simple lines of code.

First, create a format that creates dichotomous output (Missing vs Non-Missing) for all values. We need two formats, one character and one numeric, so that we can check both types of variables in our data.

```
       proc format;
         ** for numeric variables;
           value missf
                    .  = 'Missing'
                other = 'Non-Missing'
            ;
         ** for character variables;
           value $missf
                 ' ' = 'Missing'
                other = 'Non-Missing'
           ;
        run;
```

Once we have created the formats, we can use PROC FREQ to get counts of the number of Missing vs Non-Missing values for the variables.

```
        proc freq data=rawds;
               table _all_ / missing;
               format _character_ $missf. _numeric_ missf.;
        run;
```

Note that we apply the character format ($MISSF.) to all character variables by using the _CHARACTER_ in a FORMAT statement. Similarly, we apply the character format (MISSF.) to all numeric variables by using the _NUMERIC_ in a FORMAT statement. Then we use the _ALL_ in the TABLE statement to specify that a frequency table should be created for all the variables in the dataset.

Sample output looks like this:

```
                           VISDATE
                                          Cumulative    Cumulative
    VISDATE    Frequency      Percent      Frequency       Percent
   ------------------------------------------------------------------
   Missing            63         2.52             63          2.52
   Non-Missing      2436        97.48           2499        100.00
                          ERGTEST
                                          Cumulative    Cumulative
   ERGTEST     Frequency      Percent      Frequency       Percent
   ------------------------------------------------------------------
   Non-Missing      2499       100.00           2499        100.00
```

And we can clearly see that our VISDATE (or Visit Date) variable is missing on 63 records, which we will need to investigate. On the other hand, our ERGTEST (or Parameter Label) variable is non-missing for all records, so it is complete.

Additional ideas for using PROC FORMAT to validate data can be found in PharmaSUG 2005 Hands-on-Workshop paper #5 by John R. Gerlach.

HANDY USE #8: DATA CHECKING WITH THE ERROR OPTION

In general, variables have expected values or ranges. For certain variables, it is of particular importance that the values of the variables fall within expectations. As with all tasks, there are a variety of ways that we can verify values and/or flag any unexpected values. The ERROR option in PROC FORMAT provides an alternate method for verifying values.

Suppose we had two variables that we need to ensure have accurate values. One is a discrete variable, VARA, which should have a value of '01', '02', '03', or '04'. The other is a continuous variable, VARB, for which the acceptable range is positive but less than 200.

We use the following code to build two informats, one for each variable:

```
        proc format;
            invalue $errorA
                  '01','02','03','04'=_SAME_
                  Other=_ERROR_;
            invalue errorB
                  0 < - < 200 = _SAME_
                  Other=_ERROR_;
        run;
```

Note that the values we expect for the variables are assigned to _SAME_ so that they remain unchanged. Values outside the expected range ('Other') are assigned to _ERROR_. (Refer to USING THE _ERROR_ OPTION and USING THE _SAME_ OPTION above for details.) Note also that 'missing' values in our example above will fall into the 'Other' category and will thus be flagged as an ERROR. If we so desired, we could modify the above code as follows to avoid flagging missing values as errors:

```
        proc format;
            invalue $errorA
                  ' ','01','02','03','04'=_SAME_
                  Other=_ERROR_;
            invalue errorB
                  .           = _SAME_
                  0 < - < 200 = _SAME_
                  Other=_ERROR_;
        run;
```

In either case, in our analysis programming code, when we use the variable we can use a simple line of code to check each variable:

```
        _vara=input(vara, $errorA.);
        _varb=input(varb, errorB.);
```

When a value outside the established expectations is encountered in the data, an ERROR message ("NOTE: Invalid argument to function INPUT at line XXX.") is printed to the program LOG.

If these INFORMATS are stored in a centralized catalog and used for all studies, they can prove to be very useful in identifying inconsistencies among the studies. These inconsistencies can be of paramount importance during the process of ISS/ISE analyses.

This basic idea is much expanded and combined with a macro technique to automate the process in PharmaSUG 2004 Data Management paper #6 by Ronald Fehd.

HANDY USE #9: CONVERTING FROM CHARACTER MONTH TO NUMERIC MONTH

If our character dates all look like this 'DD-MON-YYYY', we can use the existing DATE9 informat to convert this character date to a SAS date, like this:

```
        _date=input(compress(char_date, '-'), date9.);
```

If our character dates all look like this 'YYYYMMDD', we can use the existing YYMMDD10. informat to convert this character date to a SAS date, like this:

```
        _date=input(char_date, yymmdd10.);
```

Occasionally our data is so messy we have no choice but to convert the dates manually. In those situations, we often find that we need to convert the character month (JAN, FEB, MAR) to a numeric value (to be read in by the MDY() function. We can use PROC FORMAT to make this conversion quickly and easily. To code to generate the format looks like this:

```
        proc format;
           value monthf
                  'JAN' = '1'
                  'FEB' = '2'
                  'MAR' = '3'
                   etc..
             ;
        run;
```

Then we use code like this to convert the month to a numeric:

```
        _date=mdy(input(put(month, monthf.), best.), day, year);
```

Or we could use an INFORMAT instead:

```
       proc format;
          invalue monthf
                 'JAN' = 1
                 'FEB' = 2
                 'MAR' = 3
                  etc..
           ;
          run;
```

And use this to create the SAS date variable:

```
       _date=mdy(input(month, monthf.), day, year);
```

For more information about working with formats and dates, refer to PharmaSUG 2002 Technical Techniques Paper #13 by Venky Chakravarthy.

CONCLUSION

Now we have a firm understanding of the basics of PROC FORMAT. We've even covered some of the not so basic options. And, perhaps more importantly, we've learned some ways that we can put PROC FORMAT to use in our everyday programming for more than just making things pretty. As you work with PROC FORMAT more and more, I hope that you will begin to see even more places where it proves to be more than just a cosmetic tool.

ADDITIONAL READING

Phillips, Stacey D., and Gary Klein, Oh No, a Zero Row: 5 Ways to Summarize Absolutely Nothing, Paper CC22, PharmaSUG 2005.
Guttadauro, Lara E.H., Use PROC FORMAT to Apply Value Specific Suffixes to Your Data, Paper CC20, PharmaSUG 2005.
Fehd, Ronald, Invalid: a Data Review Macro: Using PROC Format Option Other=Invalid to Identify and List Outliers, Paper DM06, PharmaSUG 2004
Canete, Pilita, Melanie Paules, and Shi-Tao Yeh, Using SAS Macro to Create Data Driven Format Statement for Clinical Data Flagging, Paper P04, PharmaSUG 2000.
Heaton-Wright, Lawrence, The FORMAT procedure - more than just a VALUE statement, Paper TT10, PharmaSUG 2004.
Chakravarthy, Venky, Have a Strange DATE? Create your own INFORMAT to Deal with Her, Paper TT13, PharmaSUG 2002.
Carpenter, Arthur L., Building and Using User Defined Formats, Paper TU02, PharmaSUG 2004.
Gerlach, John R., Validating Data Using SAS Formats as a Programming Tool, Paper HW05, PharmaSUG 2005.

CONTACT INFORMATION

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

Erin Christen
AYW Training & Consulting
P.O. Box 53701
Knoxville, TN 37950
Phone: (865) 455 - 6586
Email: erin@aywconsulting.com

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

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