Paper BB17

# Hidden in plain sight: my top ten underpublicized enhancements in SAS ® Versions 9.2 and 9.3

Bruce Gilsen, Federal Reserve Board, Washington, DC

## ABSTRACT

SAS ® Versions 9.2 and 9.3 contain many interesting enhancements. While the most significant enhancements have been widely publicized in online documentation, conference papers, the SAS-L internet newsgroup/listserv, and elsewhere, some smaller enhancements have received little attention. This paper reviews my ten favorite underpublicized features.

1. Eliminate observations with a unique sort key (BY groups of size one)
2. DATA step sort
3. String delimiter on input
4. String delimiter on output
5. The PRINT procedure: printing blank lines
6. Data set lists in the SET and MERGE statements
7. Append SAS log files
8. Simpler macro variable range
9. Trim leading and trailing blanks from a value stored by the SQL procedure in a single macro variable
10. Create a directory or folder in a LIBNAME statement

Also noted briefly are some interesting features in Version 9.4.

## 1. ELIMINATE OBSERVATIONS WITH A UNIQUE SORT KEY (BY GROUPS OF SIZE ONE)

Introduced in: Version 9.3

The SORT procedure NOUNIQUEKEY option eliminates observations from the output data set that have a unique sort key. That is, the output data set will not have BY groups of size one.

Optionally, the UNIQUEOUT= option specifies an output data set containing the observations eliminated by the NOUNIQUEKEY option.

Consider data set ONE, as follows.

| Obs | AA | BB | CC | DD |
|---|---|---|---|---|
| 1 | 200 | 210 | 220 | 230 |
| 2 | 200 | 220 | 220 | 230 |
| 3 | 300 | 310 | 320 | 330 |
| 4 | 1 | 2 | 3 | 4 |
| 5 | 1 | 20 | 30 | 40 |
| 6 | 1 | 2 | 3 | 4 |
| 7 | 1 | 20 | 30 | 40 |
| 8 | 1 | 20 | 30 | 40 |
| 9 | 400 | 20 | 30 | 40 |

When we sort with AA as the BY variable and use the NOUNIQUEKEY option, the observations in which AA has the values 300 and 400 are eliminated.

```
proc sort data=one nouniquekey out=one_duplicatebyvals;
  by aa;
run;
```

ONE_DUPLICATEBYVALS

| Obs | AA | BB | CC | DD |
|---|---|---|---|---|
| 1 | 1 | 2 | 3 | 4 |
| 2 | 1 | 20 | 30 | 40 |
| 3 | 1 | 2 | 3 | 4 |
| 4 | 1 | 20 | 30 | 40 |
| 5 | 1 | 20 | 30 | 40 |
| 6 | 200 | 210 | 220 | 230 |
| 7 | 200 | 220 | 220 | 230 |

If we add the option UNIQUEOUT= ONE_SINGLEBYVALS, then two output data sets are created: ONE_DUPLICATEBYVALS with the same values as above, and ONE_SINGLEBYVALS with the eliminated observations.

```
proc sort data=one nouniquekey uniqueout=one_singlebyvals out=one_duplicatebyvals;
  by aa;
run;
```

ONE_SINGLEBYVALS

| Obs | AA | BB | CC | DD |
|---|---|---|---|---|
| 1 | 300 | 310 | 320 | 330 |
| 2 | 400 | 20 | 30 | 40 |

## 2. DATA STEP SORT

Introduced in: Version 9.2

CALL SORTN (numeric) or CALL SORTC (character) sort values in a DATA step, including arrays and array elements.

DATA step sorting compares to PROC SORT as follows. If you think of a SAS data set as a rectangular box, where the rows are the observations and the columns are the variables, then

- PROC SORT sorts vertically (by observation)
- DATA step sort sorts horizontally (by variable, within the current observation only)

CALL SORTC requires all character values being sorted to have the same length; otherwise, an error occurs. CALL SORTN does not require all numeric variables being sorted to have the same length.

CALL SORTC uses your platform's default collating sequence, so on Windows, Unix and its derivatives, and OpenVMS uppercase letters are sorted before lowercase letters and on z/OS lowercase letters are sorted before uppercase letters.

Data set ONE has the following values, where N1, N2, N3, and N4 are numeric variables and C1, C2, and C3 are character variables with length 8.

| Obs | c1 | c2 | c3 | n1 | n2 | n3 | n4 |
|---|---|---|---|---|---|---|---|

```
 1     aaa    bbb    ccc     1     2     3    111
 2     eee    ddd    fff     4     6     5    222
 3     iii    ggg    hhh     9     8     7    333
```

Here is an example of DATA step sort. In each observation of data set TWO, the smallest values are in N1 and C1, the second smallest values in N2 and C2, and the largest values in N3 and C3.

```
data two;
  set one;
  call sortn (n1, n2, n3);
  call sortc (c1, c2, c3);
run;
```

```
                          TWO
Obs    c1     c2     c3     n1    n2    n3     n4
 1     aaa    bbb    ccc     1     2     3    111
 2     ddd    eee    fff     4     5     6    222
 3     ggg    hhh    iii     7     8     9    333
```

The following statements are equivalent to the above CALL SORTN and CALL SORTC statements.

```
call sortn (of n1-n3);
call sortc (of c1-c3);
```

If we create arrays containing the variables to be sorted, the following CALL SORTN and CALL SORTC statements are also equivalent to the above statements.

```
array nall (*) n1 n2 n3;
call sortn (of nall(*));

array chall (*) c1 c2 c3;
call sortc (of chall(*));
```

There is no equivalent to the DESCENDING keyword in the PROC SORT BY statement, but to sort in descending order, just reverse the order of the variables. For example, sort N1, N2, and N3 in descending order either of the following ways.

```
call sortn (n3, n2, n1);
call sortn (of n3-n1);
```

## 3. STRING DELIMITER ON INPUT

Introduced in: Version 9.2

Note that the examples in this section read data contained in the code with a DATALINES statement for readability, but apply to reading external files as well.

By default, a blank is the default delimiter (separator) for list input. The DLM= (or DELIMITER=) option on the INFILE statement provides a list of one or more delimiters to replace the blank. For example, in the following DATA step, the delimiters are 'x', 'y', and 'z'.

```
data one;
  infile datalines dlm='xyz';
  length var1 var2 $12;
  input var1 $ var2 $;
  datalines;
abcxyz123
defygxyhixyzjklmnop
;run;
```

Data set ONE has the following values. Note that the first record has consecutive delimiters.

```
Obs    var1    var2
 1     abc     123
 2     def     g
```

Until now, there was no way to specify a string delimiter; that is, a separator of 'xyz' rather than any of 'x', 'y', or 'z'. Typically, files containing string delimiters were handled with techniques such as the following.

- Pre-process the file with a SAS program, Perl script, or other software to convert the string delimiter to a single delimiter, taking care to ensure that the single delimiter is not already present in the file.
- In a DATA step, read the data values into intermediate variables and then use DATA step functions or Perl regular expressions to parse the intermediate variables.
- In a DATA step, use INPUT; to read the observations into the input buffer and parse the _INFILE_ variable.

Now, the DLMSTR= option allows you to easily read files with string delimiters, as in the following code.

```
data two;
  infile datalines dlmstr='xyz';
  length var1 var2 $12;
  input var1 $ var2 $;
  datalines;
abcxyz123
defxgxyhixyzjklmnop
;run;
```

Data set TWO has the following values. Note that for the second record, 'x' and 'xy' are treated as ordinary text, and VAR1 contains the characters preceding 'xyz'.

```
Obs    var1         var2
 1     abc          123
 2     defxgxyhi    jklmnop
```

You can specify a variable that contains the string delimiter. For example, the first record of a file could have a different string delimiter, as in the following code.

```
data three;
```

```
  if _n_=1 then delim = "xxx";  /* 1st observation */
           else delim = "xyz";  /* all other observations */
  infile datalines dlmstr=delim;
  length var1 var2 $12;
 input var1 $ var2 $;
 datalines;
abcxyzxxx123
defxgxyhixyzjklmnop
;run;
```

Data set THREE has the following values. Note that 'xyz' is treated as ordinary text in the first record.

```
Obs    var1         var2
 1     abcxyz       123
 2     defxgxyhi    jklmnop
```

The DSD option on the INFILE statement, which causes two consecutive delimiters to be treated as a missing value, can also be used with the DLMSTR option. Consider the following code.

```
data two;
  infile datalines dlmstr='xy' dsd;
  length var1 var2 var3 $12;
  input var1 $ var2 $ var3 $;
  datalines;
abcxy123xydefg
defxyghixyzjklmnop
defxyxyjklmnop
;run;
```

Data set TWO has the following values. Note that VAR2 has a missing value in the third observation.

```
Obs    var1    var2    var3
 1     abc     123     defg
 2     def     ghi     zjklmnop
 3     def             jklmnop
```

Another new INFILE statement option, DLMSOPT=, has two values that when used with DLMSTR do the following.

- "T" removes trailing blanks from the string delimiter. This is particularly useful if a variable contains the string delimiter, and the size of the string delimiter might vary.
- "I" does case-insensitive comparisons, as in the following code.

```
data four;
  infile datalines dlmstr='xyz' dlmsopt='i';
  length var1 var2 $12;
  input var1 $ var2 $;
  datalines;
```

```
abcXYZ123
defxgxyhixyzjklmnop
;run;
```

Data set FOUR has the following values. Without DLMSOPT='i', 'XYZ' in the first record would not be considered a delimiter because it does not match the case of the DLMSTR= value, and we would not get the desired result.

```
Obs     var1          var2
 1      abc           123
 2      defxgxyhi     jklmnop
```

## 4. STRING DELIMITER ON OUTPUT

Introduced in: Version 9.2

By default, a blank is the default delimiter (separator) for list output. The DLM= (or DELIMITER=) option on the FILE statement provides an alternate delimiter to replace the blank. For example, in the following DATA step, the delimiter is 'x'.

```
    /* Create some data for this example */
data one;
  input a b c;
  datalines;
1 2 3
4 5 6
;run;

data _null_;
  set one;
    /* Write output to c:\mydir\example1.txt in this example */
  file 'c:\mydir\example1.txt' dlm='x';
  put a b c;
run;
```

The file created by this example has the following two records.

```
1x2x3
4x5x6
```

Until now, there was no way to specify a string delimiter; that is, a separator of 'xyz' rather than a single character. Now, the DLMSTR= option allows you to easily specify string delimited output, as in the following code.

```
    /* Create some data for this example */
data one;
  input a b c;
  datalines;
1 2 3
4 5 6
```

```
;run;

data _null_;
  set one;
    /* Write output to c:\mydir\example2.txt in this example */
  file 'c:\mydir\example2.txt'  dlmstr='AAABBB';
  put a b c;
run;
```

The file created by this code contains the following two records.

```
1AAABBB2AAABBB3
4AAABBB5AAABBB6
```

You can specify a variable that contains the string delimiter. For example, it could be necessary to write a different string delimiter in the first record of a file than in other records, as in the following code.

```
  /* Create some data for this example */
data one;
  input a b c;
  datalines;
1 2 3
4 5 6
7 8 9
;run;

data _null_;
  set one;
  if _n_=1 then mydelim = "AAA";  /* 1st observation */
           else mydelim = "BBB";  /* all other observations */
    /* Write output to c:\mydir\example1.txt in this example */
  file 'u:\m1bfg00\example3.txt' dlmstr=mydelim;
  put a b c;
run;
```

The file created by this code contains the following three records.

```
1AAA2AAA3
4BBB5BBB6
7BBB8BBB9
```

## 5. THE PRINT PROCEDURE: PRINTING BLANK LINES

Introduced in: Version 9.2

The PROC PRINT option BLANKLINE=*n* inserts a blank line after every *n* observations. For example, in the following code, a blank line is inserted after observations 10, 20, 30, etc. are printed.

```
proc print data=one blankline=10;
run;
```

Until now, blank lines were typically displayed by adding blank observations to a data set every *n* lines, temporarily setting the MISSING= system option to a blank, and using the NOOBS option to suppress the observation numbers. Code using the older method can be simplified.

The observation count is reset at the beginning of each BY group if you use a BY statement, and includes only observations that meet the selection criteria if you use a WHERE expression.

## 6. DATA SET LISTS IN THE SET AND MERGE STATEMENTS

Introduced in: Version 9.2

Data set lists provide two new ways to read like-named data sets that are similar to features previously available in the DROP and KEEP statements: specifying a numeric range and specifying a prefix.

Data set options such as WHERE= and KEEP= are applied to a numeric range of data sets in a list by putting them in parentheses after the second name in the range or after the colon if a prefix is specified, as shown below.

The examples in this section use the SET statement, but apply identically to the MERGE statement.

**Specify a numeric range**

The first three statements read 10 data sets, A1-A10. The second and third statements apply data set options to all 10 data sets. The fourth and fifth statements read 15 data sets. The fifth statement applies different WHERE clauses to different data sets.

```
set a1-a10;
set a1-a10 (where= (x gt 100));
set a1-a10 (where= (x gt 100) keep= x y z);
set a1-a10 a20 a11-a14;
set a1-a10 (where= (x gt 100)) a20 (where= (y gt 200)) a11-a14 (where= (z gt 300));
```

All data sets in the range must exist. If data sets A1-A3 and A5 exist but A4 does not exist, the following statement generates an error. These data sets could be read with a prefix (see below).

```
set a1-a5;
```

**Specify a prefix**

The first three statements read all data sets in the library referenced by SALES that begin with jan. The second and third statements apply data set options to all the data sets. The fourth statement provides a way to read data sets A1-A3 and A5 when A4 does not exist.

```
set sales.jan: ;
set sales.jan: (where= (x gt 100));
set sales.jan: (where= (x gt 100) keep= x y z);
set a:;
```

The order that the data sets are read is determined by sorting the data set names with the same collating sequence used in PROC SORT: ASCII on Windows, Linux, and other UNIX derivatives, and EBCDIC on z/OS.

## 7. APPEND SAS LOG FILES

Introduced in: Version 9.3

The LOGPARM option allows you to append the log file from a non-interactive SAS session to an existing file. You can run multiple jobs or the same job multiple times and automatically preserve the log files from every job in one file.

The LOGPARM option is simpler than the typical ways that users save multiple log files, such as using a script that renames the existing log file or uses the system clock to name the log file for the current session. It could be particularly useful for programs that run automatically, such as Linux cron jobs.

Once you invoke SAS, no other SAS session can write to the log file associated with that session until that session ends. As such, this feature is not designed for SAS sessions running at the same time.

The example in this section uses Linux, but LOGPARM is also available on other platforms.

The following statement invokes Linux SAS, executes program sasprog1.sas, and uses mylog.log as the SAS log file, as follows.

- If mylog.log does not exist, it is created.
- If mylog.log exists, the current session is appended to the existing file.

```
sas -log "mylog.log" -logparm "open=append" sasprog1.sas
```

When you use LOGPARM, SAS writes a message similar to the following at the beginning of the SAS session. You can look for this text to determine where an appended log begins.

```
NOTE: Log file opened at Mon, 12 Mar 2012 13:18:08.468
```

## 8. SIMPLER MACRO VARIABLE RANGE

Introduced in: Version 9.3

Until now, when you copied values to a series of macro variables but did not know how many macro variables you needed, you could specify a very large number like 99999 or &SYSMAXLONG (an operating system-specific automatic macro variable containing the maximum long integer value) as the upper bound, and SAS created only as many macro variables as needed, as in the following code.

```
proc sql noprint;
  select distinct aa
  into :aa1-:aa&sysmaxlong
  from one;
%let num_values = &sqlobs;
quit;
```

The automatic macro variable SQLOBS contains the number of rows selected by the last PROC SQL statement.

In Version 9.3, you can just specify an unbounded macro variable range as follows and SAS creates as many macro variables as needed.

```
proc sql noprint;
```

```
  select distinct aa
  into :aa1-
  from one;
%let num_values = &sqlobs;
quit;
```

## 9. TRIM LEADING AND TRAILING BLANKS FROM A VALUE STORED BY THE SQL PROCEDURE IN A SINGLE MACRO VARIABLE

Introduced in: Version 9.3

When storing a value in a single macro variable, PROC SQL by default preserves leading or trailing blanks. Until now, users removed leading and trailing blanks by assigning the macro variable to itself or with other methods. In Version 9.3, the TRIMMED option trims leading and trailing blanks from values stored in a single macro variable.

Data set ONE has the following values

```
Obs    AA     BB
 1      1     11
 2      2     22
 3      5     33
 4      5     44
 5      5     55
 6      3     66
```

Here are two macro variables used in this example.

```
%let before=beforetext;
%let after=aftertext;
```

In the following code, a macro variable is created with and without the TRIMMED option.

```
proc sql noprint;
    /* Create macro variable SUM_AA1 containing the sum of variable AA */
  select sum(aa)
  into :sum_aa1
  from one;

    /* Display SUM_AA1 - it is right justified */
  %let combine1=&before&sum_aa1&after;
  %put macro variable created without trimmed option combine1=&combine1;

    /* In Version 9.3, use the TRIMMED option to remove leading and
       trailing blanks when creating a macro variable. */
  select sum(aa)
  into :sum_aa2 trimmed
  from one;
```

```
    /* Display SUM_AA2 – it has no leading or trailing blanks */
  %let combine2=&before&sum_aa2&after;
  %put macro variable created with trimmed option combine2=&combine2;
quit;
```

The SAS log includes the following.

- The first line contains SUM_AA1, which was created without the TRIMMED option; it has leading blanks.
- The second line contains SUM_AA2, which was created with the TRIMMED option; there are no leading or trailing blanks.

```
macro variable created without trimmed option combine1=beforetext      22aftertext
macro variable created with trimmed option combine2=beforetext22aftertext
```

Note that PROC SQL trims leading or trailing blanks unless you specify the NOTRIM option when either of the following is true, as in previous releases.

- The values are stored in a range of macro variables.
- The SEPARATED BY option is used to store multiple values in a single macro variable.

## 10. CREATE A DIRECTORY OR FOLDER IN A LIBNAME STATEMENT

Introduced in: Version 9.3

By default, the directory (folder) in a LIBNAME statement must already exist.

A new system option, DLCREATEDIR, allows SAS to create the directory (folder) named in a LIBNAME statement if it does not already exist. Specify this option at invocation, in a configuration file, or (as shown here) with an OPTIONS statement.

```
options dlcreatedir;run;
```

The default value of this option is NODLCREATEDIR, which prevents SAS from creating a directory.

This feature works on all platforms. It is less useful on the z/OS mainframe platform where it works with UFS directories but does not work with the traditional direct access or sequential access bound libraries employed by most users.

Here is an example using Linux SAS.

The following statement generates an error if the directory /my/home/m1xxx00/sasdir1 does not exist.

```
libname xxx '/my/home/m1xxx00/sasdir1';
```

If we specify the system option DLCREATEDIR, the LIBNAME statement successfully creates the directory.

```
options dlcreatedir;
run;
libname xxx '/my/home/m1xxx00/sasdir1';
```

Note that directories cannot be created recursively. For example, if the directory /my/home/m1xxx00/sasdir1 does not exist, the following code will not work.

```
options dlcreatedir;
```

```
run;
libname xxx '/my/home/m1xxx00/sasdir1/mydir1';
```

### Version 9.4

I have not yet had much time to work with Version 9.4, but here are a few features that have caught my interest so far, with links to papers that provide more information.

1. Using the DOSUBL function to submit SAS code that runs during a DATA step. See Langston (2013).
2. Extended data set attributes. See Olson (2013).
3. The DATA step Report Writing Interface. See Lund (2014).
4. Reading and Writing ZIP files in SAS. See Langston (2014).

## CONCLUSION

This paper reviewed some underpublicized features in SAS Versions 9.2 and 9.3. Hopefully, users will embrace these features and simplify their SAS applications.

## REFERENCES


Langston, Rick (2013), "*Submitting SAS® Code On The Side*," Proceedings of the SAS Global Forum 2013 Conference. http://support.sas.com/resources/papers/proceedings13/032-2013.pdf.

Langston, Rick (2014), "*Reading and Writing ZIP Files with SAS®,*" Proceedings of the SAS Global Forum 2014 Conference. http://support.sas.com/resources/papers/proceedings14/SAS264-2014.pdf.

Lund, Peter (2014), "*Have it Your Way: Creating Reports with the Data Step Report Writing Interface*," Proceedings of the SAS Global Forum 2014 Conference. http://support.sas.com/resources/papers/proceedings14/1362-2014.pdf.

Olson, Diane (2013), "*Developer Reveals: Extended Data Set Attributes*," Proceedings of the SAS Global Forum 2013 Conference. http://support.sas.com/resources/papers/proceedings13/135-2013.pdf.

SAS Institute Inc. (2012), "*Base SAS 9.3 Procedures Guide, Second Edition*," Cary, NC: SAS Institute Inc.

SAS Institute Inc. (2011), "SAS 9.3 Companion for UNIX Environments," Cary, NC: SAS Institute Inc.

SAS Institute Inc. (2011), "SAS 9.3 Language Reference by Name, Product, and Category," Cary, NC: SAS Institute Inc.

SAS Institute Inc. (2011), "SAS 9.3 Macro Language: Reference," Cary, NC: SAS Institute Inc.


## ACKNOWLEDGMENTS


The following people contributed extensively to the development of this paper: Heidi Markovitz and Donna Hill at the Federal Reserve Board, and Howard Schreier. Their support is greatly appreciated.


## CONTACT INFORMATION


For more information, contact the author, Bruce Gilsen, by mail at Federal Reserve Board, Mail Stop N-122, Washington, DC 20551; by e-mail at bruce.gilsen@frb.gov; or by phone at 202-452-2494.


## TRADEMARK INFORMATION