

# PROC MEANS™: More than just your average procedure

Peter R. Welbrock  
Britannia Group, Philadelphia, PA

## Abstract

PROC MEANS™ and its close relative PROC SUMMARY™ have been two of the work horse procedures that all users of SAS software learn to know intimately. Version 7 and 8 of the SAS System brought many changes to these procedures that give both the programmer and analyst new opportunities to efficiently summarize their data. This paper will give a broad overview of the MEANS procedure, starting at the seminal level with the syntax, and moving onto some of the more advanced techniques that are now available. This paper will be suitable for both the SAS beginner, since it does not assume an existing knowledge of the procedure, and any more advanced users that are looking into converting their Version 6 code to more fully exploit the opportunities available in the later releases.

This paper concentrates on the summarization of data using the MEANS procedure, rather than obtaining statistics<sup>i</sup>

## Introduction

It doesn't matter how hard one tries to avoid it, at some point the data you have is not summarized to right level. You have county level data when you want to analyze states. You have data by minutes when you actually want it by day. This usually occurs because traditionally, data is stored at its most granular level: the lowest level of detail. In many ways, this is very sensible because it does not restrict the analyst from looking at it in any way that is required. For example, if data were stored at a state level, then there would be no way to look at it at county level. That information would be lost due to the granularity of the data.

There are two trends that go against the grain: the first of these is the development of Data Warehousing. A good Data Warehouse will be designed around the understanding of the needs of the users and might therefore already contain data summarized to the level the analyst requires.

The second trend is that of placing data within an MDDB (Multi-Dimensional Database), which is in essence a series of tightly coupled pre-summarized data tables. Note that in both of these situations, the detail data (that is at the most granular level) is still available. Neither the Data Warehouse or the MDDB should restrict the analyst from looking at data in any way they require, but they are there to increase the speed in which results can be obtained, by pre-summarizing the data in a way that the analyst requires it.

Despite both the development of Data Warehouses and MDDBs, there comes a time when anyone who uses data beyond the most perfunctory reporting will still need it summarized. There are many ways this can be achieved using SAS® Software, including:

- SQL
- The data step
- PROC FREQ™
- PROC TABULATE™
- PROC REPORT™

The most often used, however is PROC MEANS because of its ratio of efficiency to ease of use. Note that PROC MEANS is used here interchangeably with PROC SUMMARY (they use the same statistical engine, sharing it with PROC FREQ). The two procedures are very similar indeed, with MEANS traditionally being used when printed output is required, as opposed to output in a data table.

The approach this paper takes to outlining the use of PROC MEANS is to use a single data source and to slowly build a single procedure, incrementally adding more options. This will allow us to step through the syntax from the simplest form, until we end up with a relatively complex example. Each incremental addition of an option will add to the functionality of the procedure. Each section of the paper will

therefore be labeled based upon the added functionality rather than the marginal syntax.

### **When would I use PROC MEANS?**

Although we have looked at the procedure in terms of summarizing data, it can do far more. There are many statistics that can be obtained from the procedure, whilst the data is being summarized. It is also possible (and a very common activity) to summarize the data into multiple levels within the same procedure step. For example, at the same time one is taking county information and summarizing it to a state level, the procedure could also be summarizing it to a country level.

*PROC MEANS can therefore be used in situations where the detail level of the data and the statistics available for the analysis variables are not compatible with the analysis needs. As previously mentioned, there are many other tools that can perform many of the tasks PROC MEANS can perform, but none available that offer the options in just a few lines of code.*

Note that it is imperative to fully understand your data. One of the major pitfalls made by users is to jump right into the analysis before really knowing what it is they are analyzing. When using PROC MEANS, it is critical to understand your categorical variables (e.g. country, state, zip) and your analysis variables in detail (e.g. number of orders, amount of orders). This should be done before summarization.

One of the perpetual discussions that occur is the efficiency of the procedure compared to other techniques that are available. This is outside the scope of this paper<sup>ii</sup>, but there are a few pointers that should be considered:

- How much extra time will it take to program the alternative and is the savings in computer usage worth it?
- Will the alternative technique be more efficient in all or just specific situations. Do I have the time to test the comparative efficiency every time I need to summarize my data?
- Does the alternative technique lead to maintenance problems in situations where the requirements are not ad-hoc?

### **Help, my data isn't giving me what I need!**

The detail data that we are going to use is based upon sales orders. The columns available to us are outlined in the following table:

Column Name	Column Type	Description
order_id	C	Order Identifier
sku	C	Product
store	C	Name of Store
year	N	4 digit year
month	N	2 digit month
quarter	N	1 digit quarter of year
store_type	C	Store Classification
total	N	\$ amount ordered
quantity	N	Number of items

**Table 1**  
**Data Table Name=sugi25.orders**

This data table, sugi25.orders contains data at an order item level. This means that there could well be multiple rows per order, each one representing a different product. For example a single order might be made up of three rows of data. The first row might be for 2 cycling helmets, the second row for 3 knee guards and the third row might be for the sales tax.

In this situation, the data is not structured in such a way to facilitate analysis on the \$ amount ordered by year, or by quarter.

### **Producing a simple summary**

A simple request from the above data would be to obtain a new data table that contains the total \$ ordered amount for each year. The syntax required to do this is:

```
proc means data=sugi.orders nway
                chartype noprint;      1)
class year;      2)
var total;      3)
output out=by_year sum=;      4)
run;
```

- 1) The *proc means* statement is standard and must be used at the start of every use of the procedure. The *data=* tells the procedure the name of the input detail data table (note

that this could be anything SAS will recognize as a SAS data set). The *nway* statement is the most complex, and restricts the number of classifications the procedure outputs to the highest level (see below). The *noprint* option informs the procedure that no printed output is required. This assumes that the user will take the output as a data set and perform additional processing. Finally, the *chartype* option is explained below.

**Notes about classifications:**

By default, MEANS will create a summary of data by every combination of columns specified in the CLASS statement. For example, if the columns *sku* and *sku\_type* were specified as class variables, then the procedure would, by default, create the following in a single output data set:

```
Total summary (_type_='00')
Sku_type (_type_='01')
Sku (_type_='10')
Sku by sku_type (_type_='11')
```

The obvious question is if all these different summary levels are output to a single table, then how is it possible to distinguish between them? This is done using a generated column in the output table called *\_type\_*. The values for the example above are shown in parentheses. In previous versions of SAS, the value for *\_type\_* was a numeric column calculated using a necessary, but rather convoluted technique where the value was incremented by one for each additional available summary level. In Version 8 and beyond, *\_type\_* can optionally (depending on using *chartype* in the MEANS statement) be a character variable containing either a '1' or '0' depending on whether the corresponding column in the CLASS statement is active in the summary level. Note that if there are more than 32 columns in the CLASS statement then *\_type\_* is automatically a character column.

It is still possible to use the old version of the *\_type\_* column, but this is probably not advisable if at all avoidable.

- 2) The *class* statement has listed after it each classification column involved within the summary. Note that these columns tend not to contain continuous variables, but discrete values. In our example, *year* is the only classification, which means that we will

have in our output only grouping by each year. If there are three years represented in our detail data, then there will only be three rows in the output data file. Note that if we had omitted the *nway* option (see 1.) then we would have the three rows plus a grouping for all the years together.

Note that in Version 8, there can be multiple class statements.

- 3) The *var* statement lists the analysis variables. In this example, only the *total* is going to be included in the output data file. Any column listed after the *var* statement has to be numeric.
- 4) The *output* statement specifies what and where the output from the procedure will be stored. The output statement can become quite complex (see examples later in the paper), but in its rawest form illustrated in our example, we are creating a new data file called *by\_year* containing the *sum* of the *total* column.

What will the *by\_year* table actually look like? The table below contains the structure of the table.

Column Name	Description
<i>year</i>	<i>year</i> value from <i>class</i> statement
<i>total</i>	<i>Total</i> from <i>var</i> statement
<i>_freq_</i>	The number of detail rows that make up the summarized grouping.
<i>_type_</i>	The level of the grouping.

**Associated Topics**

Now that we have seen the MEANS procedure in its most basic form, there are a couple of associated topics that should be addressed to fully understand how it works.

**Treatment of missing values:** by default, the MEANS procedure will remove any row from the summary if any of the classification columns contain a missing value. This behavior can be overridden using the *missing* option after the PROC MEANS statement. The *missing* option will result in a missing value being a valid grouping from a classification variable. It is easy to create misleading results from the detail data by forgetting to treat missing values in the correct way. This goes back to really understanding your data before analyzing it.

From the perspective of calculating statistics, each *var* column is treated separately. This means that if one column has a missing value on a particular row, and another column does not, then in the first instance the missing value will be omitted from the statistic calculation, but this will not affect the calculation for the second *var* column where the value is not missing.

**Treatment of SAS formats:** the MEANS procedure will take advantage of any formats that are permanently associated with a column, or any that are assigned within the procedure itself using the *formats* statement. This saves the recoding of columns before using the procedure.

Examples of using formats with the MEANS procedure are given later in this paper.

**‘But not all possible combinations of summary types are required ...’**

Let’s build on our example and now include three class variables. This will give us more possible combinations of summaries. We are going to use as our *classification* variables *year*, *store* and *sku*. In this case, the possible summary combinations are included in the table below:

Class 1	Class 2	Class 3	_type_
			‘000’
		sku	‘001’
	store		‘010’
	store	sku	‘011’
year			‘100’
year		sku	‘101’
year	store		‘110’
year	store	sku	‘111’

**Table 2  
Possible Combinations from 3 Class Variables**

We don’t, however, want all of these summaries. We only want those with *\_type\_* = ‘011’ and ‘101’. In Version 6, this could only be done by applying a *where clause* on the output data file based on the value of the *\_type\_* variable. Now, however, there is a far more elegant (and efficient) way to obtain the same results.

```
proc means data=sugi.orders missing
           chartype noprint;
class year store sku;
types (store*sku) (year*sku);
```

```
var total;
output out=by_year sum=;
run;
```

Note that the *missing* option has been added as a procedure option. This will force the procedure into using *missing* as a valid category for all of the *class* variables.

The example above illustrates the use of the *types* statement. This statement allows for the pre-selection of which interactions between the class variables are to be calculated and stored in the output data file, *by\_year*. There are many different forms of the *types* statement. Below are two examples:

```
types store* (sku year);           1)
types () sku;                       2)
```

- 1) This *types* statement is equivalent to obtaining the following interactions: *store\*sku* and *store\*year*.
- 2) This example is equivalent to obtaining the overall summary and *sku* by itself.

The *types* statement is very useful, but can sometimes be a little long-winded. There is an alternative that will apply in certain situations: the *ways* statement. This essentially restricts the output interactions to the levels included as the value of the *ways* statement. This is best illustrated by example:

```
Ways 2;                               1)
Ways 1;                               2)
```

- 1) This will give a summary of all interactions where there are two *class* variables involved i.e. *year\*store*, *year\*sku*, *store\*sku*.
- 2) This will give a summary of all interactions where there is only one *class* variable involved i.e. *year*, *store*, *sku*.

Note that it is possible to use the *types* and the *ways* statements together. Obviously, one has to be careful in this situation to ensure that the output is exactly what is required. If there is a duplication across the *ways* and the *types* requests for summary levels, then a warning will be put out to the log and any duplicates will be ignored.

### What if I want more columns in my output file than specified in the *class* or *var* statements?

It is sometimes necessary to ‘pull along’ additional columns of data into the output data file. These columns are not used as *analysis* variables, or as *classification* variables, but are nevertheless items of data that will be required in any post procedure reporting or analysis.

For example, suppose that in the output data file, we also wanted the *store\_type*. A simple way to include this in the output data file would be to use the *ID* statement. This statement informs the procedure that a column will be included in the output data file that contains the lowest value of that column from the input data file for each interaction.

Usually, the column selected for an *ID* statement will have only one value for each of the classification columns. If this is not the case, then extra care must be taken in using this statement.

For example, if we wanted *store\_type* in the output data file, then we would add the following line of code:

```
ID store_type;
```

The output data file would contain a column with the values of *store\_type*. If the summary level was *store* then this column would make sense, since it would contain the corresponding store type for each store. If the summary level were *sku* then the output *store\_type* column would have to be used with caution, since it would contain the lowest value of *store\_type* the procedure came across in summarizing the data by *sku*.

### Should the data be sorted before using the MEANS procedure?

Often, data is already sorted by the columns in the *class* statement. In this situation it is possible to use the *by* statement instead of the *class* statement. If the *by* statement is used, then it is only possible to get the *nway* interaction between class variables. This essentially limits the functionality of the procedure since it is not possible, for instance, to use the *ways* or the *types* statements.

It is not necessary to sort the file by the *class* variables. A simple index (if there is one *class* variable) or a composite index (if there are many *class* variables) could also be created.

If there is an option between using the *class* or the *by* statement, then the latter is more efficient, since it requires less memory usage. There are situations with large summaries where it is more efficient to sort the input data (removing any columns not required in the summary) and then to use *by* instead of *class*. There are, however, no simple rules for deciding between the two. There are many factors in making this decision:

- 1) The size of the data.
- 2) The amount of available memory.
- 3) The form of the data (is it already sorted?).
- 4) The required levels of summarization.

#### Efficiency

Learning about the *sumsize=* and the *memsize=* options can help in the effective use of the MEANS procedure.

Using the *by* statement is very easy. All that is required is to replace *class* with *by* in the code.

### What if formatted values of classification variables are required as summary levels?

The use of formats have already been mentioned above, but it is worth actually looking at an example. Suppose we have a user defined format as follows:

```
proc format;  
  value $ sku_fmt '0' - '3' = 'Low Sku'  
                '4' - '8' = 'Medium Sku'  
                other = 'High Sku'  
run;
```

This creates a format that we will use to re-classify the *sku* column. It will simply classify any *sku* starting with 0,1,2 or 3 as ‘Low Sku’, those starting with 4,5,6,7,8 as ‘Medium Sku’ and the rest as ‘High Sku’.

We can now use this user defined format in our MEANS procedure.

```
proc means data=sugi.orders missing  
          chartype noprint;  
class store sku;
```

```
types store*sku;
var total;
format sku $sku_fmt.;
output out=by_store_sku sum=;
run;
```

Note that the only addition to the code is the *format* statement. The output from this procedure will not contain any *sku* values, but only the formatted value of the *sku* values.

One problem with this code is that a situation might arise where for a particular summary level, there might not be a *sku* that has all three format categories. By default, this will mean that the format category not represented will not show up in the output data file (why should it?). There are circumstances where this might not be acceptable however, and Version 7 and beyond has additional functionality within the MEANS procedure to handle this (see box below).

#### Advanced Format Topic

For those seriously interested in using formats with PROC MEANS, the *preloadfmt* statement will be of use. This is useful in situations when a formatted value of a class variable is included in the output, even if there are *no* rows included in the classification.

For example, suppose a summary by *store* and *sku* (these are our *classification* variables) for *total* is desired. The classification column *sku* is to be formatted using the *sku\_fmt* format created above.

In the output, for every store, every formatted classification of *sku* should be included, *even if there is no total*. In Version 6, this would have required extensive manipulation of the data before running the procedure. This can now be achieved using the *preloadfmt* in association with the *completetypes* and *exclusive* option (see below).

The code we would use to ensure that every categorized formatted value of *sku* is included in the output is outlined below:

```
proc means data=sugi.orders missing
      chartype noprint completetypes; 1)
class store sku/preloadfmt;          2)
types store*sku;
var total;
```

```
format sku $sku_fmt.;
output out=by_store_sku sum=;
run;
```

The changes to the code are the options on the two numbered lines:

- 1) The *completetypes* option has been added. This tells the procedure to include all of the possible values whether or not they exist for a specific class interaction. In other words, in our code, for every *store*, include all the values of *sku* even if they do not exist for a particular store.

This can reduce the possible errors in using the output data file. When looking at a particular store, it will be possible to find out which skus the store did **not** have, as well as the ones it did have.

- 2) The *preloadfmt* option must be used in coordination with the *completetypes* option (see 1)), the *exclusive* option (see below) on the *class* statement or the *order=data* option (see below in the section dealing with ordering the output data file).

The *preloadfmt* if used in conjunction with the *completetypes* option will mean that all values of the formatted *class* column will occur in the output data file at all summary levels it is involved in, whether there are any rows for that value or not.

This can best be seen by the output shown in the table below:

#### Preloadfmt with Completetypes Options

Store	Skus	_Freq_	Total
Dunham's	Low Sku	5	87.2
Dunham's	Medium Sku	1	.
Dunham's	High Sku	0	.
MC Sports	Low Sku	9	532.9
MC Sports	Medium Sku	0	.
MC Sports	High Sku	0	.

**Table 3**  
Output using *completetypes* and *preloadfmt*

Note in the above table that for the Dunham's store, the formatted value of *sku*, 'High Sku' has a *\_freq\_* of zero. This means that there are no rows in the detail table that make up this row in the output data file. This row is included (as are

the final two rows for MC Sports) because of the *preloadfmt* and *completetypes* options.

#### Preloadfmt with Exclusive option

If the *preloadfmt* is used in conjunction with the *exclusive* option on the *class* statement, then the output will be restricted to those rows to which the format applies. In other words, if a sku could not be formatted, then it would be excluded from the output data file.

Thoroughly understanding the *preloadfmt* option will greatly increase the usefulness of the MEANS procedure. Although it might seem as though it is beyond the elementary use of the procedure, it is definitely an option that should be mastered.

#### A note about completetypes

This option does not have to be used in conjunction with *preloadfmt*. Without it, it will produce an output data file that has all values for a particular classification column for every other classification column.

#### **Changing the order of the data in the output.**

When producing summary data from the MEANS procedure (either in data or printed form), a specific order of the **class columns** is often required. This can, of course, be done after the fact using a procedure like SORT or SQL, but very often, it is possible to specify the sort order within the procedure itself.

There are six relatively simple ways to affect the order of the output. Four of these are implemented using the *order=* option on either the *class* statement or on the procedure statement. The final two are to use the *ascending* or *descending* option. These different options are outlined below:

#### Note on Sort Orders

Take extra care in understanding sort orders. They do vary from operating system to operating system.

*Order=data*

e.g. **class** store sku/*preloadfmt* **order=data**;

In this situation, the procedure will order the data based upon the order in which the FORMAT procedure stores them. By default, the FORMAT procedure orders values by their formatted description. It is possible to avoid this by using the *notsorted* option in the FORMAT procedure.

If the *exclusive* option is used, then only values that have a formatted value will be included. If *exclusive* is excluded, then the values that do not have a formatted value will be appended in the output after the formatted values *in the order they are encountered*.

*Order=formatted* (or *order=fmt*)

e.g. **class** store sku/ **order=formatted**;

The order of the *class* variables will be the ascending order of the formatted values.

*Order=unformatted* (or *order=unfmt*)

e.g. **class** store sku/ **order=unfmt**;

The order of the *class* variables will be the sorted order of unformatted values. This option might at first sight seem a little odd, but it is very useful in situations where you want the formatted value to be viewed in the output, but this value has little meaning beyond mere labeling. A good example of this would be dates. You might format a SAS date based so that the month shows. The output might not be what is required since if the formatted value was used to sort, then April will be first. If the unformatted value is used to sort the data, then January would be first, which is probably what is required.

*Order=freq*

e.g. **class** store sku/ **order=freq**;

The order of the class variables will be based on the *freq* variable. In other words, the observations within a particular interaction that has the highest value of *freq* would be first.

*Ascending* and *Descending*

These two options can be used on both the class statement and the procedure statement. They are

self-explanatory and specify highest-to-lowest or lowest-to-highest.

### Controlling the contents of the Output

Up until this point, our output (we have been using the *noprint* option and specifying an output data file) has been very simple. Our output has included the *class* columns, the sum of the *var* columns and the generated columns (*\_freq\_* and *\_type\_*). The *output* statement, however, has extensive uses and can become quite complex. Since this paper is an introduction to the procedure, the simpler options will be addressed, with reference to the more complex options when pertinent.

#### Multiple Class and Output Statements

Both the *class* and the *output* statements can occur multiple times within a single use of the MEANS procedure.

Using multiple *class* statements gives control over the way that *classification variables* can be treated. In Version 6, any options on the *class* statements would be applied to all *classification variables*. After Version 6, it is possible to treat different variables in different ways by using multiple *class* statements.

A similar situation arises for the *output* statement. It is now possible to have multiple *output* statements, each one producing its own output. This allows for more flexibility and reduces the number of times the procedure needs to be invoked.

The following example starts to extend the use of the *output* statement.

```
proc means data=sugi.orders missing
      chartype noprint completetypes;
class store sku/preloadfmt;
types store*sku;
var total;
format sku $sku_fmt.;
output out=by_store_sku sum=sum_total
      mean=mean_total;
run;
```

The difference in this situation is that we are using a single *var* column, but we are asking for both the *sum* and the *mean* stored in the output as columns called *sum\_total* and *mean\_total* respectively.

Using this technique, it is possible to create output that has multiple statistics for multiple columns. Taking this a step further by adding an additional *var* column:

```
proc means data=sugi.orders missing
      chartype noprint completetypes;
class store sku/preloadfmt;
types store*sku;
var total quantity;
format sku $sku_fmt.;
output out=by_store_sku sum=sum_total
      sum_quantity
      mean=mean_total
      mean_quantity;
run;
```

In this situation, we have four calculated columns in our output, based on the two *var* columns *total* and *quantity*. For each of these we have the *sum* and the *mean*.

Both of these two examples have illustrated a simple way to obtain statistics and to explicitly name the output columns. It is not necessary, however, to explicitly specify the statistics required:

```
proc means data=sugi.orders missing
      chartype noprint completetypes;
class store sku/preloadfmt;
types store*sku;
var total;
format sku $sku_fmt.;
output out=by_store_sku;
run;
```

The above snippet of code excludes explicit reference to both the output statistics and their corresponding column names. What would happen here is that the output would include a row for each interaction between *store* and *sku* for the following statistics: mean, std, n, min and max. A new column would be created called *\_stat\_* that would contain the name of the statistic. The column *total* would contain the values of the statistic.

The process of explicitly stating the names of the output variables can be tedious. In Version 6, this was a major chore if there were a large number of *var* columns and a large number of statistics required. After Version 6, the

*autoname* option alleviates this tedious procedure.

```
proc means data=sugi.orders missing
  chartype noprint ;
class store sku;
types store*sku;
var total quantity;
format sku $sku_fmt.;
output out=by_store_sku n= mean= range=
      max= min= median=
      / autoname ;
run;
```

This example uses the *autoname* option to automatically create the names of the output columns. In this situation, there are only two *var* variables and six statistics. This requires naming twelve output columns. The *autoname* option will create the output column name by appending the statistic keyword to the end of the *var* column name. So in the example above the first two columns names for the statistics will be: *sku\_n* and *sku\_mean*.

Note that if the *var* column label is required, as opposed to the name, then the *autolabel* option should be substituted.

### Subsetting the output

In Version 6, subsetting the output was largely dependent on two different techniques:

- Eliminating unwanted data before the procedure runs.
- Eliminating unwanted data during the creation of the output.

The first of these two options should be a standard technique not just for the MEANS procedure, but for any situation where processing takes place. It can be performed by putting a *where clause* on the incoming data, a *where clause* within the procedure or, if there is an efficient opportunity, in code before the procedure.

Examples of these three techniques are illustrated in the simple code below.

```
proc sort data=sugi.orders out=work.orders
  nodupkey; 1)
by store sku year month;
run;
```

```
proc means
  data=orders(where=(sku='4'))
  missing; 2)
by store sku;
types store*sku;
var total quantity;
format sku $sku_fmt.;
output out=by_store_sku(where=(total > 999))
  sum=; 3)
run;
```

- 1) The data is pre-processed to remove any duplicate values. Note that this could not be done within the MEANS procedure.
- 2) A *where clause* is applied to the data as it comes into the procedure. Note that this will be more effective when the data is large if the incoming data is indexed by the columns referenced within the *where clause*.
- 3) The data is subsetting on the way out of the procedure, using calculated results.

It would have been possible to use the subsetting in 2) within the SORT procedure. This decision would be based on whether the *work.orders* data file would be used elsewhere.

There are now more options available than just using the *where clause*. A slightly more complex example is as follows. In this situation, we want to obtain the three skus by store that have the greatest orders (total).

```
proc means data=sugi.orders missing
  chartype noprint;
class store ;
types () store ;
var total ;
format total dollar10.2;
output out=by_top_total sum= mean=
  idgroup( max(total) id[3] (total sku)=) / 1)
  autolabel autoname;
label total='Total $';
run;
```

Note that most of the code is very basic. The *total* column is the analysis variable and there is only one *classification* variable, which is *store*. The big difference is with the *output* statement. The basis of the change is the use of the *idgroup* option, which extends the functionality of the *id* statement we examined earlier (see line 1)).

The *idgroup* option gives the MEANS procedure the ability to output a data file that contains

extreme values. In our example, we are obtaining the following:

**max(total)**

Will give the maximum values of the *total* column for each outputted row, after the procedure has run.

**Id[3]**

Will restrict the output to the top three *max* values of the *total* column

**(total sku)=**

Will inform the procedure that in finding the top three *total* values for each row in the output, include in that output the values of the *total* and *sku* columns. In other words, there will be three *sku* and three *total* values in the output data file. This will mean that we will know the three skus that have the greatest order amount for each of the rows in the output data file and the values of those amounts.

One of the dangers of using the *idgroup* option is the complexity of naming conventions. It is possible to logically have duplicate variable names in the output file. For this reason, it is more than advisable to use the *autolabel* and/or *autoname* options.

Another issue with the *idgroup* option is that of ties. What happens if 10 of the skus have identical order amounts (*total*). In this situation, the procedure will select the skus to output based upon observation number. Just to reiterate an earlier point, this illustrates how important it is to really know the data being used.

The use of *idgroup* is one of the more complex pieces of using the MEANS procedure, but it offers a set of additional tools that can greatly increase the usefulness of the procedure.

### **Conclusion**

The MEANS procedure is one of the most important available. It is worth spending the time to become adept at using the myriad of options available. The time invested in familiarizing yourself with the procedure will pay off many times over. It doesn't matter if SAS is being used for elementary or sophisticated uses since this procedure always has its place.

In many ways, it is now more important than ever to fully get to grips with PROC MEANS since it is strategically a very important piece of

the SAS software. It is central to the creation of HOLAP MDDBs as well as in the data warehousing process, where success is based upon giving the users data in the form closest to their business requirements.

In this paper, the basics of the procedure have been covered. There are many more options available, all of which are covered in the online documentation.

To learn more, a good place to start is to study each of the samples given in both the online documentation and within the Technical Support page on the SAS web site.

---

### **References:**

<sup>i</sup> For a beginning introduction to using PROC MEANS for creating statistics, see: Delwiche, Lora D. and Susan J. Slaughter, *The Little SAS® Book, Second Edition*, Cary, NC: SAS Institute Inc., 1998. Page 172.

<sup>ii</sup> For efficiency techniques see: Bob Virgile, *Efficiency: Improving the Performance of Your SAS Applications*, Cary, NC: SAS Institute Inc., 1998.

### **Author:**

Peter R. Welbrock  
Strategic Information Systems, Inc.  
Philadelphia, PA  
610 668 6417  
[welbrock@erols.com](mailto:welbrock@erols.com)

Author of '*Strategic Data Warehousing Principles Using SAS® Software*', Cary, NC: SAS Institute Inc.,