Paper 041 - 2018

Using Memory Resident Hash Tables to Manage Your Sparse Lookups

Arthur L. Carpenter, California Occidental Consultants, Anchorage, Alaska

ABSTRACT

When dynamically controlling an application or process through the use of SAS® macros, it is often advantageous to utilize a combination of metadata control files and list processing techniques. Typically the control file is read once into a macro variable list. But what if your information source is a large table and instead of accessing it sequentially you need to first look up specific information based on some supplied criteria? It would be very useful to be able to load the table into memory once and then only access those portions needed at a given point in time. It would be even better if you could do this without creating any macro variable lists. Learn how this can be done using a combination of hash tables, FCMP functions, and the macro language to create memory resident lookup tables.

KEYWORDS

Metadata, Macro Language, Hash Object, PROC FCMP, Control Files, %SYSCALL, %SYSFUNC, functions, routines, RESOLVE

INTRODUCTION

The use of control files to drive a macro process (Rosenbloom and Carpenter, 2015) is commonly implemented by creating a list of macro variables. The items in this list are generally then processed sequentially one at a time (Fehd and Carpenter, 2007). This approach is both powerful and efficient, however it is not necessarily the best approach for all situations.

Consider the situation where you are given or you derive a coded value, perhaps a customer identifier, and that you need to use that value to retrieve information stored in a large data set. This is a table lookup scenario and there are numerous ways to retrieve the necessary information (Carpenter, 2015 and Horstman, 2017). When the table is large and the lookup criteria complicated often one of the best approaches is the use of a hash table. This is especially the case when the lookup is being performed in the DATA step.

The use of the DATA step is fine, but what if you are writing a macro application where the DATA step cannot be used? Or what if you are going to need to use the same lookup table in more than one DATA step? Can we still take advantage of the use of hash tables, and can we do this by loading the hash table only once – even across DATA steps? Unfortunately the answer is not unilaterally yes, however starting with SAS9.4M4 the technique can be practical in some usages as the lookup table can now be retained in memory without reloading the data into the hash table for each use. This is accomplished by defining the hash table within a FCMP function and by then calling the FCMP function from within the macro language.

The hash object will be declared in an FCMP routine (Liu, 2017) and that routine will then be either be called in a DATA step or by the macro language. Although there will be some review of PROC FCMP in the examples that follow, readers unfamiliar with the FCMP procedure are encouraged to first gain a basic understanding using other sources, such as Carpenter (2013) or Eberhardt (2009). Similarly there will be a short review of hash object syntax, however the reader new to hash objects should first review Schacherer (2015).

The Lookup Table

In this presentation we have a very big data set that we will need to use as our look up table (for demonstration purposes the original 19 observations in the SASHELP.CLASS data set have been expanded to 1,900,000 distinct names). Because sorting is often an issue with actual lookup operations, this data set has been 'unsorted' on the index variable (NAME) by sorting it on a random number (RAND).

```
data big(keep=name age height weight rand);
   length name $15;
   set sashelp.class(rename=(name=oldname));
   do i = 1 to 100000;
      rand = ranuni(123456789);
      name=cats(oldname,i);
      output big;
   end;
   run;
proc sort data=big;
   by rand;
   run;
```

The index variable is NAME, which will take on values such as 'Alice4567', and values to be retrieved include the variables AGE, HEIGHT, and WEIGHT. The lookup objective will be to return the height, weight, and age values for a given name.

Defining a Hash Object

Hash objects are defined (declared) using a declaration block which is initiated with a DECLARE statement, and then further defined through the use of methods. During the declaration of the method, the hash object is named and the key(s) and data field(s) are stated.

```
declare hash class(dataset:'work.big'); ❶
   rc=class.definekey('name'); ❷
   rc=class.definedata('name','height','weight', 'age'); ❸
   rc=class.definedone(); ❹
```

❶ The DECLARE statement states that we are defining a hash object named CLASS. The DATASET: constructor tells SAS to load the data set WORK.BIG into the hash object.

❷ The DEFINEKEY method names the key variable(s) that will be used as the index to access this hash object. Although this example has a single key variable, you can have multiple variables.

❸ The DEFINEDATA method names the variable(s) that are to be stored in the hash object. The variable RAND exists in the data set WORK.BIG, but it is not being included in the hash table, therefore it will not be available to be returned when the hash table is used.

❹ The DEFINEDONE method closes the definition of the hash object.

Defining a FCMP Function

PROC FCMP is used to create user defined functions. The details of the syntax has been described in a number of papers Carpenter (2013, 2018), Eberhardt (2009), and Secosky (2007). FCMP can be used to define both functions and routines. Functions return a single value, while routines can return any number of values. Because table lookup scenarios tend to need to return multiple values, this discussion will concentrate on the generation of FCMP routines that are accessed through the use of the CALL statement (or the %SYSCALL statement when accessing through the macro language). In PROC FCMP, routines are defined using a SUBROUTINE statement, which starts a block of code ending with an ENDSUB statement.

```
proc fcmp outlib=work.functions.hash; ❶
   subroutine GetStats(name $, height,weight,age) ; ❷
      outargs height, weight, age; ❸
      declare hash class(dataset:'work.big'); ❹
         rc=class.definekey('name');
         rc=class.definedata('name','height','weight', 'age');
         rc=class.definedone();
      rc=class.find(); ❺
   endsub; ❻
   run;
```

❶ The OUTLIB= option identifies where the function definition will be stored. WORK.FUNCTIONS will be a specialized data set in the WORK library.
❷ The routine is to be named GETSTATS, and will have a single character argument (NAME) and three numeric arguments.
❸ The OUTARGS statement is used to identify the arguments that are to be returned by the routine.
❹ The hash object CLASS is defined within the subroutine definition.
❺ The FIND() method is used to retrieve a value from the hash table. A value for NAME will be passed into the subroutine when it is called (NAME is the first argument of the routine). Because it has been defined as a key variable, the FIND method will then use the value of NAME as the index to look up and retrieve the values of the data variables from the hash object.
❻ Close the definition of the GETSTATS routine.

It is important to note that the data table (WORK.BIG) is not loaded into the hash object when the subroutine is defined (compiled) by FCMP. The data set does not even need to exist at this point in time.

USING THE ROUTINE

Once defined, the FCMP routine can be utilized wherever you normally use functions and routines, however you will need to tell SAS where to look for your user defined functions. In the PROC FCMP step the OUTLIB= option identified a specialized type of data set to hold our function definitions. Now we will use the CMPLIB= system option to point back to that location or data set.

```
options cmplib=(work.functions);
```

In The Data Step

The hash object can be used to perform a lookup by calling the subroutine GETSTATS. In this simple DATA step the values of height, weight, and age are returned for two different names (❷, ❺).

```
data lookup1;
   retain height weight age .; ❶
   name='Alfred23456'; ❷
   call getstats(name,height,weight,age); ❸
   output; ❹
   name='Alice98765'; ❺
   call getstats(name,height,weight,age);
   output; ❹
   run;
```

❶ The variables to be returned must be defined on the PDV before the routine is called. This is a necessary requirement for any routine that returns values and is not because our routine uses a hash object.
❷ The name that is to be looked up is specified.
❸ The routine GETSTATS is called using the CALL statement. The returned values are added to the PDV.
❹ Because he GETSTATS routine adds the values of the returned variables to the PDV, the observation is now ready to be written to the new data set.
❺ The process is repeated for a second name.

In The Macro Language

Functions and routines created by PROC FCMP can automatically be utilized by the macro language through the use the %SYSFUNC function and the %SYSCALL statement; %SYSFUNC is used to execute functions and %SYSCALL to execute routines.

You will need to take the same basic steps when using the routine with the macro language as you did in the DATA step. Before the routine can be called, the arguments, including the values to be returned, need to be defined. Here the %CALLGETSTATS macro is passed the name to lookup through the &NAME parameter.

```
%macro callgetstats(name=Alfred1);
   %local name height weight age ; ➊
   %let height=.; ➋
   %let weight=.;
   %let age=.;
   %syscall getstats(name,height,weight,age); ➌
   %put &=name &=height &=weight &=age; ➍
%mend callgetstats;
%callgetstats(name=Barbara98765) ➎
```

➊ Forcing the macro variables to be local is not required.
➋ The routine's arguments must be initialized. Since they are assumed to be numeric, they could not be simply initialized to null with the %LOCAL or %GLOBAL statement.
➌ The routine is called using the %SYSCALL statement. As is typical with the %SYSCALL statement, notice that the arguments to the routine are assumed to be names of macro variable names, and are specified without the ampersand.
➍ Here a %PUT is used to show that the values have been successfully retrieved. More typically a process, like a macro call, that utilizes these values would go here.
➎ The desired name to look up is specified in the call to the %CALLGETSTATS macro.

The SAS Log shows that the routine has successfully retrieved the information for Barbara98765.

```
161  %callgetstats(name=Barbara98765)
NAME=Barbara98765 HEIGHT=65.3 WEIGHT=98 AGE=13
```

PERFORMANCE ENHANCEMENTS

The use of hash objects in FCMP function has been available for quite some time, however with SAS9.4M4 enhancements were made that allow us to make the hash table memory resident. To verify whether or not the hash table is stored in memory, a number of tests were performed to determine when the hash table is loaded into memory and whether or not it is reloaded on subsequent calls both from within and across DATA steps and from one usage by the macro language to the next.

All the testing benchmarks shown below are machine and operating system (OS) dependent, however the highlighted relationships should remain consistent regardless of machine configuration or OS.

Because the PROC FCMP step was executed before the data set WORK.BIG was created, it was easily verified that the data set, which will be loaded into the hash table, does not need to exist when the routine is compiled. The routine was established without a problem, therefore the WORK.BIG is clearly not loaded into the hash table during the execution of the PROC FCMP step.

Testing – Within and Across DATA Steps

The first test was to determine the time needed to lookup a single set of values in a DATA step. On the machine used for the testing (Windows 7 64 bit running SAS9.4M4) this step executed in approximately 4.8 seconds. Most of this time is needed to load the hash table when the routine is first called.

```
data lookup;
   retain height weight age .;
   name='Alfred23456';
   call getstats(name,height,weight,age);
   output;
   run;
```

We can show that the hash table is not being held in memory across DATA steps by deleting WORK.BIG and then rerunning this step. The second step fails when the routine is called and is unable to load the WORK.BIG data set.

The hash table is however held in memory within a DATA step. When we make multiple calls to the routine in the same DATA step, the hash table is loaded only once and is available for instant use on subsequent calls. We can show this by calling the routine twice within a single step. The data set LOOKUP1 has two observations and still executes in under 5 seconds. In fact a DO loop that randomly requested the data associated with 200 names still executed in about the same time as the request for the values associated with two names.

```
data lookup1;
   retain height weight age .;
   name='Alfred23456';
   call getstats(name,height,weight,age);
   output;
   name='Alice98765';
   call getstats(name,height,weight,age);
   output;
   run;
```

When requests were made in two different DATA steps the same savings were not realized across DATA steps. This confirms that the hash table was not saved in memory across the DATA step boundary. The first call to the GETSTATS routine within a DATA step causes the hash table to be reloaded.

The two DATA steps shown here randomly select one set of values from the hash table for each name on SASHELP.CLASS. The time to execute the first DATA step is essentially the same as the time required for the second DATA step. This indicates that the hash table is not maintained in memory across the DATA step boundaries and must therefore be reloaded.

This is a limitation of these techniques and further work needs to be done (see the section below on EXTENSIONS and AREAS FOR FUTURE DEVELOPMENT).

```
data lookup1;
   retain height weight age . done 0;
   length namenum $15;
   do while(done=0);
      set sashelp.class(keep=name)
          end=done;
      rand = ceil(ranuni(987654)*100000);
      namenum = cats(name,rand);
      call getstats(namenum,height,weight,age);
      output lookup1;
   end;
   run;

data lookup2;
   retain height weight age . done 0;
   length namenum $15;
   do while(done=0);
      set sashelp.class(keep=name)
          end=done;
      rand = ceil(ranuni(987654)*100000);
      namenum = cats(name,rand);
      call getstats(namenum,height,weight,age);
      output lookup2;
   end;
   run;
```

Repeated Calls in the Macro Language

Unlike in the DATA step where the hash table is only maintained within memory within a DATA step, once the routine is called using %SYSCALL the hash table remains in memory throughout the remainder of the SAS session (or until deleted).

```
%macro callsub(nam=Alfred, n=1);
   %local name height weight age ;
   %let height=.;
   %let weight=.;
   %let age=.;
   %let strt=%sysfunc(datetime()); ❶
   %do i = 1 %to &n;
      %let rand = %sysevalf(%sysfunc(ranuni(987654))*100000,ceil); ❷
      %let name = &nam&rand; ❸
      %syscall getstats(name,height,weight,age); ❹
      /* process utilizing returned values goes here */
      %*put &=name &=age;
   %end;
   %let stop=%sysfunc(datetime()); ❺
   %let time= %sysevalf(&stop-&strt);
   %put &=n &=time;
%mend callsub;
%callsub(nam=Barbara, n=1) ❻
%callsub(nam=Barbara, n=100) ❼
```

This is demonstrated by the macro %CALLSUB, which can be used to call the GETSTATS routine any number of times.

❶ The retrieval start time is noted.

❷ A random number between 1 and 100,000 is generated.

❸ The name to be retrieved is determined.

❹ The GETSTATS routine is called using %SYSCALL.

❺ The time after retrieval is noted and the elapsed time is calculated.

❻ The first time %CALLSUB is executed (with the retrieval of a single name), the macro executes in just under 4 seconds.

❼ However on the second execution which retrieves 100 names, the execution takes less than .2 seconds. Clearly the hash table has been loaded once and is held in memory across executions of the macro %CALLSUB.

TAKING ADVANTAGE OF MEMORY RESIDENT HASH TABLES ACROSS DATA STEPS

Loading a hash table once and then reusing it multiple times can have tremendous advantages. Once the function is called from within the macro language the hash table is stored in memory, however as has already been demonstrated, the hash table is reloaded when the routine is called in successive DATA steps. If we can determine a way to circumvent the reloading of the hash table when working across DATA steps, we can still take advantage of this savings. Since we know that the hash table is not reloaded when used within the macro language, let’s explore some potential macro language solutions.

Using %SYSCALL Instead of CALL in the DATA Step

At first blush one potential solution might seem to be to replace the DATA step’s CALL statement with the %SYSCALL and thus take advantage of how the macro language holds the table in memory. Unfortunately this is not in any way a viable solution. Macro language elements are executed before the DATA step is even compiled, and therefore the %SYSCALL cannot be executed during the DATA’s execution phase. The %SYSCALL statement cannot be used as a substitute for the CALL statement.

Using CALL EXECUTE

Unlike the %SYSCALL statement, the CALL EXECUTE routine is executable and could be used to make successive macro calls, and successive macro calls can load and reuse a memory resident hash table. In the DATA _NULL_ step shown here the macro %CALLSUB, which was shown above, is called once for each observation in the incoming data set. Because the %CALLSUB macro calls the %GETSTATS function through the use of the %SYSCALL statement the hash table is stored in memory and reused for successive calls. This is true even if additional DATA steps also access the function.

```
data _null_;
   retain height weight age .;
   length name $15;
   set sashelp.class(keep=name);
   rand = ceil(ranuni(987654)*100000);
   name = cats(name,rand);
   call execute(cats('%callsub(nam=',name,',n=1)'));
   run;
```

While this technique allows us to access the memory resident hash table across DATA steps, it has limitations. What if we need to use the values retrieved from the lookup table (the student stats) within the current DATA step. This cannot be done using CALL EXECUTE, as the macro calls are actually executed after the DATA step terminates.

Using the DOSUBL Function

Unlike the CALL EXECUTE routine, which causes macro calls to be executed after the DATA step has completed, the DOSUBL function can be used to execute macro code <u>during</u> DATA step execution. Although the use of DOSUBL seems like it may solve our problem, this approach actually results in a substantially poorer performance.

```
%macro tstdata(val);
   %put &=val;
   %local height weight age ; ❶
   %let height=.;
   %let weight=.;
   %let age=.;
   data new;
      set sashelp.class(keep=name rename=(name=n)); ❷
      retain height weight age .; ❸
      length name $15;
      rand = ceil(ranuni(987654)*100000);
      name = cats(n,rand); ❹
      call symputx('name',name,'l');
      rc=dosubl('%syscall getstats(name,height,weight,age)'); ❺
      age=symgetn('age'); ❻
      height=symgetn('height');
      weight=symgetn('weight');
      output new;
      run;

      . . . . the second DATA step similar to the first is not shown . . . .

%mend tstdata;
```

❶ The macro variables are instantiated.
❷ The student's name is read from the incoming data set.
❸ The variables to hold the retrieve statistics are initiated.
❹ The name to be retrieved is generated.
❺ The statistics are retrieved from the hash table using %SYSCALL and the DOSUBL function, and then they are loaded into macro variables.
❻ The macro variable values returned by the %SYSCALL GETSTATS are retrieved and loaded into the DATA step variables.

Although in previous examples %SYSCALL GETSTATS utilized a memory resident version of the hash table, the hash table is not preserved across DATA step boundaries even when using it with DOSUBL. The DOSUBL function executes within its own domain and since its domain is tied to a specific DATA step, it cannot transcend DATA step boundaries. Performance is further degraded by the additional overhead associated with the DOSUBL and SYMGETN functions.

Using the RESOLVE Function

(Although I wish that I had thought of this approach myself, it was actually suggested to me by both Bart Jablonski and Thomas Billings)

The RESOLVE function allows the resolution and execution of macro language elements from within the DATA step. Typically it is used to resolve macro variables, however its uses are much more varied. As is the case throughout the DATA step, macro language elements enclosed in single quotes are not resolved during DATA step compilation. When a macro language element, such as a macro call, is placed in single quotes and passed as an argument to the RESOLVE function, the resolution (and subsequent macro execution) must take place when the RESOLVE function executes. Using this approach we can control a macro's execution during the execution of a DATA step.

First the %CALLGETSTATS macro shown earlier is modified to return a string that we can parse. This version of %CALLGETSTATS is now a macro function, into which we pass a name and the macro returns the items of interest as a single string, with values separated by an exclamation point (!).

```
%macro callgetstats(name=Alfred1);
 %local name height weight age ;
 %let height=.;
 %let weight=.;
 %let age=.;
 %syscall getstats(name,height,weight,age);
%bquote(&name)!%bquote(&height)!%bquote(&weight)!%bquote(&age) ❼
%mend callgetstats;
```

❼ For Barbara98765, the %CALLGETSTATS macro returns: `Barbara98765!65.3!98!13`

This new version of %CALLGETSTATS can now be used in a DATA step in conjunction with the RESOLVE function. Before the RESOLVE function ❽ can execute, the macro call to %CALLGETSTATS must be resolved and executed. That this happens during DATA step execution is exactly what we need to have happen. Because %CALLGETSTATS performs the %SYSCALL on the FCMP routine, the hash table is loaded and held in memory. The string returned by the RESOLVE function must then be parsed, but this is ultimately a very small price to pay. Subsequent DATA step executions utilize the same stored hash table, consequently it does not need to be loaded a second time.

```
data a1;
length str $ 1000;
name = 'Barbara98765';
str = resolve('%callgetstats(name='||strip(name)||')'); ❽
height = scan(str, 2, "!");
weight = scan(str, 3, "!");
age    = scan(str, 4, "!");
run;
```

Although it is mostly cosmetic, we could go one step further and disguise the use of the resolve function by placing it in its own routine. The routine RETRIEVEHWA, shown here, contains the call to the RESOLVE routine and parses the results into the three numeric variables of interest.

```
proc fcmp outlib=work.functions.hash;
    subroutine retrieveHWA(name $, height, weight, age);
      outargs height, weight, age;
      length str $200;
      str = resolve('%callgetstats(name='||strip(name)||')');
      height = input(scan(str, 2, "!"),best12.);
      weight = input(scan(str, 3, "!"),best12.);
      age    = input(scan(str, 4, "!"),best12.);
   endsub;
run;
```

The user of the RETRIEVEHWA routine does not even need to know that they are using a macro or the RESOLVE function.

The DATA step that uses this routine is now straightforward and does not require any special knowledge on the part of the user other than how to use a routine that returns values.

```
data a1;
name = 'Barbara98765';
height=.;
weight=.;
age=.;
call retrieveHWA(name, height, weight, age);
put (_all_) (=);
run;
```

This approach is a bit awkward, but it effectively gives us a good work around for holding a hash table in memory across DATA steps.

EXTENSIONS AND AREAS FOR FUTURE DEVELOPMENT

Because the hash table is saved in memory when called by the macro language this gives us a very nice tool, but, other than the workaround just mentioned, I found the limitation of the hash table not being saved across DATA steps to be disheartening. As was shown above several attempts were made, with limited success, to try to overcome this limitation.

In this short section are some suggestions of alternative approaches. My hope is that a reader of this paper who has more experience with the following techniques can expand on them with more success, and increase the utility of memory resident hash objects.

Using the SASFILE Statement

The SASFILE statement can be used to load a data set into memory. Although a hash table is technically a variable on the Program Data Vector, PDV, it cannot be saved to the data set itself.

```
sasfile work.big open;
```

When working across DATA steps, saving the large data set to memory will decrease the load time on subsequent reads, however the savings are not nearly as significant as one would hope.

Using PROC DS2

PROC DS2 allows what is essentially multiple DATA steps in a single PROC DS2 step, and although it seems that there are some memory resident possibilities here, this does not seem to be the case. DS2 does not currently (SAS9.4M4) allow you to call FCMP hash table routines from within a DS2 step, however the hash table can instead be loaded into a DS2 package. Even so the hash table does not remain memory resident across DATA steps within the DS2 step. This seems to be the case when using DS2 threads as well.

Using the FEDSQL

Mark Jordan (the SAS Jedi) posited that PROC FEDSQL might offer a partial solution. It is possible to load a data table into memory so that its repeated access is quicker. However here again, the solution is incomplete as the hash table itself is not held in memory.

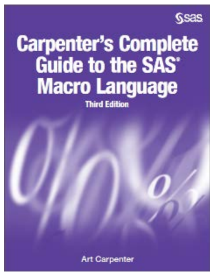
CONCLUSION

Through the use of FCMP functions and routines it is possible to store hash tables in memory. Information can then be retrieved from the hash table through function and routine calls. When used in conjunction with the macro language, the hash table can remain in memory across the execution of separate macro executions. This memory residence can have a huge performance advantage when the hash table is reused throughout a program.

Unfortunately, without using the workaround solution mentioned above, current limitations prevent the hash table from being stored across DATA step boundaries and additional research and potentially software development is needed to overcome this limitation.

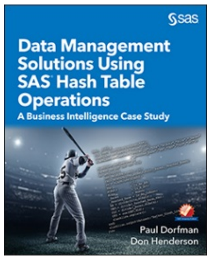
REFERENCES

Carpenter, Arthur L., 2013, "Table Lookup Techniques: From the Basics to the Innovative". Presented at the Wisconsin Illinois User Conference 2013. Also presented at MISUG 2014, PharmaSUG 2014, WUSS 2014, and 2015 SAS Global Forum. http://www.pharmasug.org/proceedings/2014/AD/PharmaSUG-2014-AD03.pdf



Carpenter, Arthur L., 2013, "Using Proc FCMP to the Fullest: Getting Started and Doing More", Presented at 2013 and 2018 SAS Global Forum Conferences, PharmaSUG, and other regional conferences. http://support.sas.com/resources/papers/proceedings13/139-2013.pdf

Carpenter, Art, 2016, *Carpenter's Complete Guide to the SAS® Macro Language, Third Edition*, SAS Institute Inc, Cary, NC. http://support.sas.com/publishing/authors/carpenter.html


sas
Data Management Solutions Using SAS Hash Table Operations
A Business Intelligence Case Study
Paul Dorfman
Don Henderson

Dorfman, Paul and Don Henderson, 2018, *Data Management Solutions Using Hash Table Operations: A Business Intelligence Case Study*, SAS Institute Inc, Cary, NC. https://www.sas.com/store/books/categories/examples/data- management-solutions-using-sas-hash-table-operations-a-business-intelligence-case-study/prodBK_69153_en.html

Eberhardt, Peter , 2009, "A Cup of Coffee and Proc FCMP: I Cannot Function Without Them", presented at the 2009 SAS Global Forum Conference. http://support.sas.com/resources/papers/proceedings09/147-2009.pdf

Fehd, Ronald and Art Carpenter, 2007, "List Processing Basics: Creating and Using Lists of Macro Variables" by Ronald Fehd and Art Carpenter which was presented at the 2007 SAS Global Forum (Paper 113-2007). The discussion of the paper looks at different approaches used in the automation of programs by using various kinds of macro variable lists. This paper appears in proceedings of a number of conferences, including: SASGF(2007), WUSS (2008), MWSUG (2009), SESUG (2009).
http://www.sascommunity.org/wiki/List_Processing_Basics_Creating_and_Using_Lists_of_Macro_Variables

Horstman, Joshua, 2017, "Beyond IF THEN ELSE: Techniques for Conditional Execution of SAS® Code", presented at the SouthEast SAS Users Conference. Paper BB-144-2017. http://analytics.ncsu.edu/sesug/2017/SESUG2017_Paper-144_Final_PDF.pdf

Liu, Qing, 2017, "Streamline Table Lookup by Embedding HASH in FCMP", PharmaSUG China 2017 - Paper 19 https://www.pharmasug.org/proceedings/china2017/AD/PharmaSUG-China-2017-AD02.pdf

Rosenbloom, Mary F. O. and Carpenter, Arthur L., 2015, "Are You a Control Freak? Control Your Programs – Don't Let Them Control You!", presented at the SAS Global Forum 2015 Conference in Dallas, Texas (paper 2220-2015). http://support.sas.com/resources/papers/proceedings15/2220-2015.pdf

Schacherer, Chris, 2015, "Introduction to SAS® Hash Objects", presented at SAS Global Forum, paper 3024-2015. https://support.sas.com/resources/papers/proceedings15/3024-2015.pdf

Secosky, Jason, 2007, "User-Written DATA Step Functions", presented at the 2007 SAS Global Forum Conference. http://www2.sas.com/proceedings/forum2007/008-2007.pdf

ABOUT THE AUTHOR

Art Carpenter is a SAS Certified Advanced Professional Programmer and his publications list includes; five books and numerous papers and posters presented at SAS Global Forum, SUGI, PharmaSUG, WUSS, and other regional conferences. Art has been using SAS since 1977 and has served in various leadership positions in local, regional, and international user groups.

CONTACT INFORMATION

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

Arthur L. Carpenter
California Occidental Consultants
10606 Ketch Circle
Anchorage, AK 99515

(907) 865-9167
art@caloxy.com
www.caloxy.com

Certified Clinical Trials Programmer Using SAS®9

Certified Advanced Programmer for SAS®9

View Art Carpenter's paper presentations page at:
http://www.sascommunity.org/wiki/Presentations:ArtCarpenter_Papers_and_Presentations

TRADEMARK REFRENCES