

An Analysis of SAS® Coding Techniques

Carl R. Haske, STATPROBE, Inc.

Abstract

All SAS programmers develop a bag of tricks that they can draw from to solve complex programming problems. A good set of reusable programming tools will include useful utilities that can be applied repeatedly in virtually any application. In the SAS language, like any other language or development platform, there are usually several ways to code a solution to a particular problem. For example, a method to determine the existence of a data file can be designed a number of different ways including the use of PROC Contents, PROC SQL with the DICTIONARY tables, or using SCL functions. Each method has its own advantages by measure of readability, maintainability, conciseness, and performance. This paper examines a number of different methods for a set of SAS utilities that are coded using an assortment of techniques. Finally, benchmark test results are compiled to compare the performance of the methods and the methods are contrasted for conciseness and maintainability.

Introduction

In any programming environment, the ability of a program to get information about an object in the environment is paramount. This ability is what allows a program to be flexible, data driven, and have logical branching dependent on the various attributes of an object. For example, an iterative statistical process may check a data set for the presence of a race or gender variable to initiate additional analyses on sub-populations. If the variable exists, then analysis on the sub-populations occurs, otherwise no analysis is performed.

In SAS, the objects we work with include logical libraries, physical directories, operating systems, data sets, variables, catalogs, and programs. Getting information about these objects, such as object existence and object attributes, helps add power to the processing and scope of a program.

Prior to the advent of DICTIONARY tables and %Sysfunc, programmers developed several clever techniques to get at some of this information. For example, something as innocuous as the number of variables in a data set required fairly cumbersome and inefficient coding to obtain. Many seasoned SAS programmers remember lamenting the absence of the seemingly obvious features that a complete programming system should possess, particularly system level processes. Thankfully, many of these shortcomings have been addressed in later versions of SAS as the system becomes more complete and self contained. The SAS Institute is notable for listening to and responding to requests from its users.

In this paper, we discuss some techniques to code various utility operations. We start with some legacy type coding techniques, that are doubtless still in use at several companies, and show how to use the DICTIONARY and %Sysfunc to write programs that are more efficient, understandable, and provide more elegant solutions to programming coding utilities.

Utilities

Application 1: Data File Existence

This section discusses three distinct methods that determine the existence of a SAS data file in a SAS session. This is a standard type of knowledge that a program can use for logical branching dependent on file existence. Version 6.12 of the SAS system offers the %Sysfunc, which provides an interface to a powerful set of SCL functions. This has streamlined the way in which you can determine file existence by allowing use of the EXIST function.

Prior to %Sysfunc, you could use the Dictionary tables and develop a macro that returned information about the existence of a data set. Before Dictionary tables became available in version 6.08, various methods from using PROC Contents to shelling out to a system function call could be used.

Here we have three methods, beginning with an application of PROC Contents, then an application of Dictionary tables, and finally an application of %Sysfunc. An analysis of these methods shows that %Sysfunc is far and away the best method to accomplish this task.

All methods are implemented as a macro and the interface to each macro is similar. All macros begin with some statements to validate proper usage with respect to the macro parameters. In addition, each macro returns the existence state in a macro parameter. Possible return values for the state are -1, 0, and 1. The value -1 indicates that an error in processing occurred, 0 represents non-existence, and 1 denotes existence.

Method 1: PROC Contents

The core code in the method works by executing a PROC contents on the input library and saving the results to a temporary data set which is then scanned searching specifically for the data set name that is passed in as a parameter. A data _null_ step is used for the scanning process. The exist macro variable is initialized to -1, then set to 0 if processing is successful, and finally set to 1 if the data set is found. A final clean-up step is required to delete the temporary data set that was created. The code appears in listing 1.

Listing 1. PROC Contents method

```
%Macro dsexist(lib=WORK,
               memtype=DATA,
               data=,
               exist=dsexist);

%Global &exist;
%Let &exist=-1;

%If &data= %then %do;
  %Put WARNING: DATA is blank.
  Macro will stop processing;
  %Goto FINISH;
%End;

%Let data=%upcase(&data);

Proc contents data=&lib._all_mt=&memtype
  noprint out=the_data (keep=memname);

%Let &exist=0;
Data _null_;
  Set the_data
  (where=
    (upcase(memname)="&data"));
  call symput("&exist",1);
  stop;

Proc datasets library=work nobs nolist;
  delete the_data;
  quit;

%FINISH;
%Mend dsexist;
```

Method 2: Dictionary Tables

The early validation steps in this method are similar to the previous method. The core code uses a single PROC SQL statement to load the *exist* macro variable with 0 or 1, based on whether the data set is found in the DICTIONARY table called *Tables*. No clean-up steps are necessary in this method. The code appears in listing 2.

Listing 2. DICTIONARY Table method

```
%Macro tblexist(lib=WORK,
               memtype=DATA,
               tblname=,
               exist=tblexist);

%Global &exist;
%Let &exist=-1;

%If &tblname= %then %do;
  %Put WARNING: TBLNAME is blank.
  Macro will stop processing;
  %Goto FINISH;
%End;

%Let lib=%upcase(&lib);
%Let memtype=%upcase(&memtype);
%Let tblname=%upcase(&tblname);

%If &memtype ne DATA or
  &memtype ne VIEW %then
  %Let memtype=DATA;

Proc sql noprint;
  Select count(memname) into: &exist
  From dictionary.tables
  where libname in("&lib") and
         memtype="&memtype" and
         memname="&tblname";

Quit;

%FINISH;
%Mend tblexist;
```

Method 3: %SYSFUNC

The final method is the most elegant. This method contains a single line of code using %Sysfunc for the core processing. In fact, it is probably more desirable to call %Sysfunc in-line in your program rather than bury it within a macro. One advantage of this type of macro construction is that it could be used to replace a later version of an existing macro that uses slower code and it is easy to make backward compatible. This would make applications that call the existing macro process faster. The code is in listing 3.

Listing 3. %Sysfunc method

```
%Macro d_exist(lib=work,
               data=,
               exist=d_exist);

%Global &exist;
%Let &exist=-1;

%If &data= %then %do;
  %Put WARNING: DATA is blank.
  Macro will stop processing;
  %Goto FINISH;
%End;

%let &exist=
  %sysfunc(exist(&lib..&data));

%FINISH;
%Mend d_exist;
```

Method Comparison

Benchmark testing was done on a computer with a 233 MHz Pentium processor and 32 MB RAM running SAS 6.12. The macros were tested on the SASHELP library that contained 61 data sets. Benchmarks include *Lines of core code*, *Total lines of code*, and *Time to process*. The Lines of core code benchmark includes a count of only those code statements that are essential to obtain the information. The benchmark for total lines of code includes a count of essential lines and additional macro coding such as error checking. Time to process is gathered from the SAS log. Programmers that were polled overwhelmingly preferred the %sysfunc method, however there was some sentiment that the Dictionary table method was elegant and interesting.

Table 1. Benchmarks for data file existence

Method	Macro Name	Lines of core code	Total Lines of code	Time to Process (sec)
PROC Contents	DSEXIST	5	19	5.71
Dictionary Tables	TBLEXIST	2	16	.88
%sysfunc	D_EXIST	1	10	*

*Too brief to measure

The benchmark measures give a clear indication that %Sysfunc is the method of choice. The conciseness of code and speed of processing time support the use of this method for determining the existence of a data file.

Application 2: Data Set Attributes

Older techniques for getting data set attributes involve a derivative of the PROC Contents Method described in the previous section. For example, consider the case of obtaining the number of variables in a data set. Basically, the technique is to generate a PROC Contents on the input data set and output the results to a temporary data set. This temporary data set is scanned to find maximum value of the variable number (varnum) field which represents the number of variables in a data set.

This method can not be programmed without running PROC Contents and at least one additional data step procedure or PROC. This process is inherently inefficient and it provides little return for all the CPU processing required, just a small piece of information about the data set.

In this section, we look at two better techniques for retrieving this type of information. Rather than presenting macro tools that perform this function, we concentrate on the core coding that is needed. A macro application with error checking is easily developed as in the previous section and is left as an exercise for the reader. In the two applications presented here, the attribute is identified as a parameter. Table 2 shows the various attributes that can be obtained with these methods. To compile this table, we assume that only the Dictionary table TABLES is used and that PROC Contents results are output to a data set.

Table 2. Data Set Attributes

Description	Attribute Name: Dictionary tables method	Attribute Name: %Sysfunc method
Data set label	MEMLABEL	MEMLABEL
Data set type	MEMTYPE	TYPEMEM
Creation date	CRDATE	CRDATE
Modification date	MODATE	MODATE
Number of observations	NOBS	NOBS
Observation Length	OBSLEN	*
Number of variables	NVAR	*
Type of password protection	PROTECT	PROTECT
Compression Routine	COMPRESS	COMPRESS
Reuse Space	REUSE	REUSE
Buffer Size	BUFSIZE	N/A
Number of deleted observations	DELOBS	DELOBS
Type of indexes	INDXTYPE	IDXUSAGE
Sorted Status	N/A	SORTED
Sort Variable	N/A	SORTEDBY
Number of indexes	N/A	IDXCOUNT
SAS Engine	N/A	ENGINE

*Results are not immediately available without additional processing

N/A = Not available

Method 1: Dictionary Tables

First, using the DICTIONARY tables, it is very easy to determine the number of variables in a data set. The code for this is presented in listing 4. The code is exhibited as if it were embedded within a macro and it uses macro expressions for four parameters as follows:

- name - the name of the macro variable to store the data set attribute
- attrib - identifies the desired attribute
- lib - the libref for the data set
- table - the name of the data file
- memtype - the file type of data or view

Listing 4. Using DICTIONARY tables to retrieve a data set attribute

```
Proc sql noprint;
  Select &attrib into : &name
  From dictionary.tables
  where libname="&lib" and
         memname="&table" and
         memtype="&memtype";
Quit;
```

The DICTIONARY table used is *tables*. The table attribute (&attrib) identifies the desired attribute to return to the calling routing, for example, *nvar* for the number of variables in a data set. The correct row of the table is selected using a where statement to identify the library (&lib), data set (&table), and member type (&memtype). The value for the attribute is stored in a macro variable represented by the macro parameter *name*. This code is very straightforward, brief, and elegant, especially in comparison to older methods used to get attributes of a data set. It uses a single SQL select statement to get the desired information requiring minimal CPU and data processing.

Method 2: %Sysfunc

The analogous code that uses %Sysfunc is in listing 5. This code is also very brief, however it clocks in at a total of three statements. All functions used are SCL functions and they are accessed via %Sysfunc. The first line opens the data set using the *open* function. The open function returns a data set identifier or handle that is stored in the macro expression *dsid* and allows subsequent code to reference the data set. The second line uses the *attrn* function passing the *dsid* handle as the first parameter and the attribute specified by &attrib as the second parameter. The attrn function returns the value of the attribute. Finally, the third line uses the *close* function to close the data set. The close function is necessary to make the data set available for processing by other SAS sessions.

Listing 5. Using %Sysfunc to get the number of variables in a data set

```
%let dsid=%sysfunc(open(&lib..&table));
%let &name=%sysfunc(attrn(&dsid,&attrib));
%let rc=%sysfunc(close(&dsid));
```

Comparison

Both of these methods are fast, straightforward, and easy to comprehend. Overall, neither of these methods has a huge advantage over the other. Most of the commonly needed data set attributes are available from both methods. The benchmarks for these methods were calculated by averaging the processing time for retrieving the four attributes label, creation date, and number of observations.

Table 3. Benchmarks for number of variables

Method	Lines of core code	Time to Process (sec)
DICTIONARY Tables	2	.17
%Sysfunc calls	3	*

* Too brief to measure

Application 3: List of variables in a data set

Data-driven SAS applications often need to build a list of data set variables for processing purposes and this section discusses techniques for building such a list. The two techniques described here closely parallel two of the three techniques used earlier to determine the existence of a data set. The first method is PROC Contents based and the second uses DICTIONARY tables. For simplicity, each method is presented without error checking code so that we may focus on the core function of the method.

Let us state the requirements for each method simply as: For a given data set, the method must produce six arrays of macro variables for each of variable attributes name, length, type, label, format, and informat.

Method 1: PROC Contents

This method runs PROC Contents on the given data set and stores the results in a temporary output data set. The number of variables in the temporary data set is counted using PROC SQL and the dictionary tables. Global macro array variables are declared for each of the six variable attributes name (&name), length (&lenname), type (&typname), label (&label), format (&fmtname), and informat (infmtname). Finally, a data _null_ procedure reads the temporary data set as input and loads the variable attribute values in to the global macro arrays. This method is presented as a macro with parameters:

- lib - the data set library
- data - the data set name
- count - the macro to store the number of variables in the data set

Listing 6. PROC Contents method

```
Proc contents data=&lib..&data out=templist noprint;

Proc sql noprint;
  Select nobs into : &count
    From dictionary.tables
    where memname="TEMPLIST";
Quit;

%Do i=1 %to &&count;
  %Global &name&i &lennam&i &typnam&i
    &label&i &fmrtnam&i &infnam&i;
%End;

Data _null_;
  Set templist end=eof;
  call symput("&name"||trim(left(_n_)),name);
  call symput("&typnam"||trim(left(_n_)),upcase(type));
  call symput("&label"||trim(left(_n_)),label);
  call symput("&fmrtnam"||trim(left(_n_)),format);
  call symput("&infnam"||trim(left(_n_)),informat);
Run;
```

Method 2: Dictionary Tables

The primary difference between this method and the previous method is that PROC SQL and the dictionary tables are used to create a view that stores the variable attributes. This view is used to load the global macro arrays using the same technique as the previous method. This method is presented as a macro with parameters:

- lib - the data set library
- table - the table name
- memtype - the table type, either DATA or VIEW
- count - the macro to store the number of variables in the data set

Listing 7. Dictionary Tables method

```
Proc sql noprint;
  Select nvar into : &count
    From dictionary.tables
    where libname="&lib" and
      memname="&table" and
      memtype="&memtype";
Create view ____cl as
  Select name, type, length, label, format, informat
    From dictionary.columns
    where libname in("&lib") and
      memname="&table" and
      memtype="&memtype";
Quit;

%Do i=1 %to &&count;
  %Global &name&i &lennam&i &typnam&i
    &label&i &fmrtnam&i &infnam&i;
%End;

Data _null_;
  Set ____cl;
  call symput("&name"||trim(left(_n_)), name);
  call symput("&lennam"||trim(left(_n_)), length);
  call symput("&typnam"||trim(left(_n_)), upcase(type));
  call symput("&label"||trim(left(_n_)), label);
  call symput("&fmrtnam"||trim(left(_n_)), format);
  call symput("&infnam"||trim(left(_n_)), informat);
Run;
```

Application 4: Resolving a date in the future

In STATPROBE's data processing environment, a large number of SAS data sets are processed in the Windows environment. In some cases, data sets have modification dates in the future because some users accidentally set their clock to a future date. When this happens, no other user can open and access the data set because SAS gives a "Datetime in the future" error. The IT group found a clever solution to the problem. Basically the solution involves setting a computer's time to a date in the future after the date on the offending data set, opening the data set using some method like FSEDIT, and, while the data set is open, setting the clock to the current date, then closing the data set. This will reset the date on the data set and make it accessible to all users.

The requirements to automate this process are to batch process all data sets in a specified data library using a specific date in the future that is greater than all the data set modification dates. Two methods were developed to accomplish this and they are essentially similar, however the first method uses standard SAS code and the second method uses SCL code.

Method 1: Standard code

The code for this method appears in listing 8. The first step scans the input library, &lib, and creates a view that contains the name of all the data sets. The next step loads the data sets into a macro array. Finally, this macro array is used to process each data set. The key in this processing is to shell to the operating system and set the date to the future. The data set is opened using the open function. While the data set is open, the operating system is used again to set the date to the current date. Finally the data set is closed. Any problematic data set is corrected.

Listing 8. Standard code

```
%macro fixdall(lib=,fdate=,curdate=);
Proc sql;
  Create view temp as
  Select memname
  From dictionary.tables
  Where libname="&upcase(&lib)";
Quit;

Data _null_;
  Set temp end=eof;
  Call symput('ds'||trim(left(_n_)),trim(left(memname)));
  If eof then call symput('numds',trim(left(_n_)));
Run;

%do k=1 %to &numds;
  x "date &fdate";
  %let dsid=%sysfunc(open("&lib..&ds&k"));
  x "date &curdate";
  %sysfunc(close(&dsid));
%end;
%Mend fixdall;
```

Method 2: SCL Method

This method is implemented in SCL. The basic steps are identical to the previous method. An advantage of this method is processing speed. A disadvantage is that it is harder to use. The programmer that uses this method should know how to interface data step code with SCL code. The best advantage of this method is the ability to easily integrate it into a graphical user interface.

Listing 9. SCL Method

```
Method library path future current $;
  libcl=loadclass('sashelp.fsp.library.class');
  libobj=instance(libcl);
  call send(libobj,'_assign_',library,path);

  memlist=makelist();
  call send(libobj,'_get_members_',
    memlist,"memtype='DATA'");

  do i=1 to listlen(memlist);
    rc=system('date '||future);
    dsid=open(getnitemc(getiteml(memlist,i),
      'LIBNAME')||'|'.'||
      getnitemc(getiteml(memlist,i),
      'MEMNAME'),'U');
    rc=system('date '||current);
    rc=close(dsid);
  end;
Endmethod;
```

CONCLUSION

In this paper, we presented four utilities that were implemented using several techniques. It is typical that any programming problem has more than one solution. In some cases, there is a best solution, however in many cases multiple solutions are essentially equivalent with the only differentiation being in the specific application.

This paper, along with presenting some interesting techniques, attempts to stimulate the thought process of the programmer into considering many approaches to problem resolution taking into account processing speed, usability, and maintainability.

ACKNOWLEDGMENTS

Thanks to Joe Huettelman for suggesting the application to correct the date in the future problem. Thanks to Annette Pickard for assistance in preparing this paper.

SAS and SAS/AF are registered trademarks of SAS Institute Inc. in the USA and other countries. ® indicates USA registration.

AUTHOR'S ADDRESS

Carl R. Haske, Ph.D.
STATPROBE, Inc.
3885 Research Park Drive
Ann Arbor, Michigan 48108
(734) 769-5000 x115
E-mail: chaske@statprobe.com