Building Data-Driven Applications Using the SAS® Applications System: Selected Techniques

Staff of SAS Consulting Services Inc., Rockville, MD

ABSTRACT

This tutorial describes a system architecture by which an application can be maintained and enhanced through changes to data and parameter files. An application designed this way allows for more flexibility and lower maintenance than one in which code is written to perform specific functions. The generalized code in a data-driven system performs under the direction of parameters, allowing any changes to be implemented by altering values in parameter files instead of changing the source code. Examples include such cases as

- adding new reports
- changing or reordering menu items
- controlling user access and security.

Sample programs that make use of SAS/AF® and SAS/FSP® software using Screen Control Language(SCL), the SQL procedure, and the SAS macro facility are presented and described in detail.

Because the data-driven approach can be adapted to a wide variety of data-processing tasks involving varying degrees of complexity, the techniques presented here merit attention as models which may suggest solutions to many system design problems.

INTRODUCTION

The primary objective of application developers is to build flexible and easily maintained systems. One good way to accomplish this is through the use of a data-driven architecture, for which the SAS System is ideally suited.

Central to the data-driven approach is the idea that a system can be designed to allow modification or expansion by changing items in data sets instead of altering program source code. Data-driven features can be utilized at all levels of an application, from formatting items in a report to making additional functional components available to the user. With a system that follows such design principles, a user with no programming knowledge can readily customize and enhance the application.

The following sections develop progressively the data-driven concept, illustrating topics with applicable code fragments and short listings of data sets. "SAMPLE DATA" introduces a sample database which is used throughout the text, initiating the data-driven discussion with a simple use of PROC FORMAT. "REPORTING" demonstrates how SAS tools (particularly PROC SQL in conjunction with the macro facility) can be used to aggregate data for a report, applying user-specified report definitions which are maintained in a data set. This topic is further developed in "MACRO LOOPING", automating the generation of code with the use of macro looping. "MAINTAINING MENUS" shows how the tools available in SAS/AF and SAS/FSP software can be used to build a data-driven user interface which can be customized and enhanced by the end-user. Together, these topics present a comprehensive overview of data-driven techniques that can be used in almost any SAS application.

Throughout this tutorial, we refer to data files that drive the application as *parameter files*, because values in these data sets are passed as parameters to program entries, SAS macro routines, or SAS procedures, thereby controlling the execution of the application.

SAMPLE DATA

As an example, a simplified database has been created for GIZMO Manufacturing. The database consists of two SAS data libraries; one contains the actual application data and the other contains the parameter files that drive the system. The application data library has only one SAS data set, INVENTRY, representing the current inventory, and containing the variables QUANTITY, PRODUCT, and WHSECODE. The variable QUANTITY contains the number of boxes of the product. The PRODUCT is the product code for the product. The WHSECODE is the code for the warehouse where the product is stored while awaiting distribution. Table 1 shows sample data from this file.

Table 1 Sample Data from the INVENTRY Data Set

Product Code	Warehouse Code	Quantity
A12	20	109
A21	10	43
A30	30	37
B39	50	17
B48	20	128
B57	30	3
B66	20	13
C75	20	149
C84	50	150
C93	10	188

The second library contains the parameter files that the system uses to perform certain tasks. The parameter file library contains the PRODUCTS file, consisting of product code, a product description, the cost per box, and the price per box. A second file, the WAREHSE data set, consists of the warehouse codes and the warehouse description. These data sets are shown in Tables 2 and 3. These parameter files are used to create CNTLIN data sets for use in a PROC FORMAT step to output the necessary formats for reporting. Other parameter files will be discussed later.

Table 2 Sample Data from the PRODUCTS Data Set

Product	Product Description	Unit Cost	Unit Price
A12	STRANGE WIDGET	1.62	1.89
A21	MEDIUM WIDGET	0.20	0.23
A30	SMALL WIDGET	1.14	1.33
B39	ROUND GADGET	0.32	0.37
B48	TRIANGULAR GADGET	1.90	2.22
B57	CHARTREUSE GADGET	1.96	2.29
B66	FUCHSIA GADGET	0.92	1.07
C75	LAVENDER THINGAMAJIG	0.68	0.79
C84	ORDINARY THINGAMAJIG	1.68	1.96
C93	SPECIAL THINGAMAJIG	0.88	1.03

Table 3 Sample Data from the WAREHSE Data Set

Warehouse Description	Warehouse Code
NORTHEAST	10
MID ATLANTIC	20
ROCKY MOUNTAIN	30
MID WEST	40
SOUTHERN	50

The following code is an example of a PROC FORMAT step using a CNTLIN data set.

```
data makefmt;
   set parms.warehse;
   rename whsecode=start;
   rename desc=label;
   type = 'C';
   fmtname="$warehse";
run;

proc format cntlin=work.makefmt;
run;
```

REPORTING

Producing reports is an important aspect of any data processing project, and there are many tools available within the SAS System to accomplish this. These tools include the PRINT, REPORT, TABULATE, and SQL procedures, among others. The usefulness of these tools can be expanded through the use of data-driven architecture. Parameter files are used to define the reports.

Consider the following examples using the INVENTRY data described previously:

1. The manager of the Widget Division needs a report to help him manage a reorganization of the product lines. He is assuming responsibility for the "STRANGE" line and will no longer be responsible for CHARTREUSE WIDGETS. The report should contain QUANTITY, the total cost of the inventory (the sum of UNITCOST), the total value of the inventory (the sum of UNITPRC), and will contain four lines of numbers. The first line is the total number of widgets (the PRODUCT code begins with an A). The second line is the total number of STRANGE GADGETS (B12) and STRANGE THINGAMAJIGS (C12). The third line is the number of CHARTREUSE WIDGETS (A57). And, finally, the last line is the total of lines 1 and 2 minus line 3.

2. Another manager needs a similar report. Line 1 is the total number of GADGETS and THINGAMAJIGS, line 2 is the number of CHARTREUSE WIDGETS, and line 3 is the total of these two lines.

There are many other managers who have similar reporting needs and we do not want to write, test, and maintain separate report programs for each request.

Instead of creating a separate report program for each of these cases, we can define a single reporting program that uses parameter files to define the data aggregation. The data set in Table 4, PARMS.REPTLINE, defines all the aggregations for both reports described above. The variable REPTID identifies the report to which each LINE belongs; the variable LINE specifies how to aggregate the lines together and control the order of the lines in the report; the variable PRODUCT indicates the PRODUCT codes to include; and, the variable FACTOR is used to control addition, subtraction, discounts, and so on.

Table 4 Report Definition Parameter File

REPTID	LINE	PRODUCT	FACTOR
WIDGET1	1	A	1
WIDGET1	2	B12	1
WIDGET1	2	C12	1
WIDGET1	3	A57	1
WIDGET1	4	A	1
WIDGET1	4	B12	1
WIDGET1	4	C12	1
WIDGET1	4	A57	−1
GADTHNG	1	B	1
GADTHNG	1	C	1
GADTHNG	2	A57	1
GADTHNG	3	B	1
GADTHNG	3	C	1
GADTHNG	3	A57	1

The following SQL code can be used to produce our report. Our report program is implemented as a macro so that we can pass in the desired REPTID as a parameter and readily use the code elsewhere in our application. We can also pass in the report title using the macro parameter TITLE.

```
%macro report(reptid=,title=);~
%*********************************************************************;
%*Sample report program that uses a parameter file and PROC SQL to  *;
%*do the data aggregation and produce the actual report              *;
%*********************************************************************;

proc  sql;
   title "&title";
   select line,
          sum(quantity*factor) as quantity,
          sum(quantity*factor*unitcost) as inv_cost format=dollar10.2,
          sum(quantity*factor*unitprc) as inv_valu format=dollar10.2
   from parms.reptline r,
        widget.inventry i,
        parms.products p
   where r.product = substr(i.product,1,length(r.product))
        and i.product = p.product
        and r.reptid = "&reptid"
   group by line;
quit;
%mend report;
```

The macro call, `%report(reptid=WIDGET1,title=Sample Report for WIDGET1)`, produces the output in Output 1.

Sample Report for WIDGET1

LINE	QUANTITY	INV_COST	INV_VALU
1	8105	$6,894.22	$8,039.45
2	1522	$875.58	$1,018.80
3	973	$486.50	$564.34
4	8654	$7,283.30	$8,493.91

Output 1 Sample Report for WIDGET1

The output shown in Output 1 does not have meaningful labels for the report rows. By adding another parameter file, PARMS.LINETEXT, containing one observation for each line in each report, we can provide a description for each report line (See Table 5).

Table 5 Parameter File of Line Definition Text

REPTID	LINE	TEXT
WIDGET1	1	All Widgets
WIDGET1	2	"Other" Strange
WIDGET1	3	Chartreuse Widgets
WIDGET1	4	My New Products

The updated macro, with changes underlined, follows:

```
%macro report(reptid=,title=);
%*********************************************************************;
%*Sample report program that uses a parameter file and PROC SQL to  *;
%*do the data aggregation and produce the actual report             *;
%*********************************************************************;

proc sql;
   title "&title";
   select text,
          sum(quantity*factor) as quantity,
          sum(quantity*factor*unitcost) as inv_cost format=dollar10.2,
          sum(quantity*factor*unitprc) as inv_valu format=dollar10.2
   from parms.reptline r,
        widget.inventry i,
        parms.products p,
        parms.linetext l
   where r.product = substr(i.product,1,length(r.product))
         and i.product = p.product
         and r.reptid = "&reptid"
         and r.line = l.line
   group by r.line, l.text;
quit;
%mend  report;
```

The macro call `%report(reptid=WIDGET1,title=Sample Report for WIDGET1)` now produces the output in Output 2.

Sample Report for WIDGET1

TEXT	QUANTITY	INV_COST	INV_VALU
All Widgets	8105	$6,894.22	$8,039.45
"Other" Strange	1522	$875.58	$1,018.80
Chartreuse Widgets	973	$486.50	$564.34
My New Products	8654	$7,283.30	$8,493.91

Output 2 Sample Report for WIDGET1

Note the use of the SUBSTR function in the SQL WHERE clause. Although the current version of the SQL procedure does not support truncated comparisons using the SAS =: operator, we can obtain the equivalent results by truncating the longer expression to the length of the shorter one before comparing them.

We have developed a mechanism that can produce any report of this nature. All that is required is a straightforward SQL program and two parameter files. The SQL procedure is a very powerful data management and reporting facility that can be used to add even more capabilities to our reporting application. For example, the logic shown here uses a simple aggregation based on a selection using a single key variable. We could use a variety of other comparison and WHERE operators that the SQL procedure supports to do other types of selection or aggregation, such as using multiple key variables to define the aggregation. This method can be used iteratively so that the results of a step can be used for later aggregations (for example, once line 2 is defined, it can be used directly in the definition of line 4). The level of sophistication of the method used to select and aggregate data is almost unlimited. We could use the SQL procedure to create data sets or views that are passed to other procedures or DATA steps to produce the desired output. Refer to the *SAS Guide to the SQL Procedure: Usage and Reference, Version 6, First Edition* for more information on the capabilities of PROC SQL.

MACRO LOOPING

Using the macro language to generate data-dependent SAS code is also a valuable technique. In our example, we need to run the REPORT macro for each value of REPTID in the report definition parameter file described in the preceding section. We need a data-driven approach so that as new report definitions are added to the file, the new reports will be generated automatically.

We first need to create a list of unique REPTID values. We will do this by reading an SQL view of unique REPTIDs from the report definition parameter file, PARMS.REPTLINE. The SQL code to create such a view follows. Since the view is permanently stored, the SQL code only needs to be run one time, not every time the reports are generated.

```
proc sql;
   create view parms.runrpts as
   select distinct(reptid) from parms.reptline;
quit;
```

For each observation read from this view, we create a macro variable with a standard prefix for the name and a uniquely numbered suffix. For example, the DATA step that follows creates a series of macro variables, RPT1 through RPT*n* where *n* is the number of REPTID values to be processed. The value of each macro variable is a different REPTID value.

After we run the DATA step shown in the macro RUNRPTS, we have a list of REPTID values that we need to process. Next we want to call our report macro once for each REPTID. We do this by calling the macro inside the iterative %DO loop. Note that the statements %DO and %END can only be used inside of a macro.

```
%macro runrpts;


 /* Code to create list of macro variables RPT1-RPTn.                */

 data _null_;

   /* read view: unique list of report ID values                     */
   set parms.runrpts end=lastrec;

   /* Create one macro var for each obs. For information on SYMPUT   */
   /* see SAS Guide to Macro Processing, Version 6, Second Edition   */
   /* page 165 "SYMPUT Routine."                                     */
   call symput('RPT' || left(put(_n_,5.)), reptid);

   /* at end create a macro var that tells how many we created       */
   if lastrec then call symput('N', left(put(_n_,5.)) );

 run;
```

```
  /* Loop through the list of report IDs and call the report macro  */
  /* once for each value.                                           */

  /* The loop index I is a macro variable and should be made local  */
  %local i;

  %do i=1 %to &n;
     /* Note the reference below is scanned twice during resolution.*/
     /* The first time && resolves to &, RPT is treated as constant */
     /* text, and &I resolves to the current index value. The second*/
     /* time we are actually resolviog &RPTn to a report ID value.  */
     /* See SAS Guide to Macro Processing, Version 6, Second Edition*/
     /* page 36 "Scanning Macro Variable References."               */

     %report(reptid= &&rpt&i, title="Report for &&RPT&i" );

  %end;

%mend runrpts;
```

Any time we want to run all of the reports defined by the report definition parameter file, we invoke the RUNRPTS macro. The macro will use the RUNRPTS view to get a list of what reports to run, create a macro variable for each report ID, and then call the REPORT macro for each report ID by referencing these macro variables.

MAINTAINING MENUS

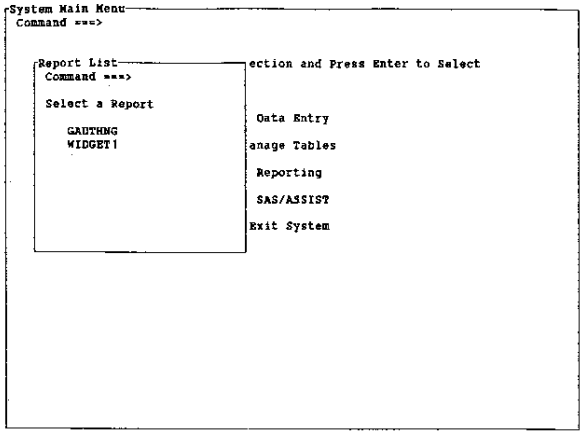
The application should also allow the end-users to interactively select any specific report from a menu. In addition, users should be able to update the report list without having to modify program code. The SAS System provides many tools to create selection lists, such as MENU and PROGRAM entries in SAS/AF software and functions within Screen Control Language (SCL), available in both SAS/AF and SAS/FSP software.

Using MENU entries, the programmer builds menus that provide a list of selections to users. Chosen selections are accessed through additional MENU entries and PROGRAM entries that navigate the user through the application. A MENU entry is a hard-coded option to process a selection. This requires a developer to modify the application in order to add or delete selections.

PROGRAM entries are a powerful and versatile tool for building menus. SCL code directs the execution of the user's choice. There are several ways that a PROGRAM entry can be used as a menu, utilizing functions available in SCL and the attributes associated with the entry. For example, the BLOCK function can be used to display a menu (such as those in SAS/ASSIST® software) containing choice blocks with labels identifying the functional pieces of the application. However, the BLOCK function is limited to only 12 choices. Alternatively, a dynamic menu can be created by using an extended table, a special type of PROGRAM entry that allows the developer to repeat a set of fields defined in the display window. In a typical application using extended tables, SCL is written to process each row of the table, corresponding to an observation from a data set. This use of extended tables as menus is critical to data-driven applications development.

Selecting a Report

An extended table allows the user the flexibility of presenting a list of selections which are values within a data set (See Display 1).



Display 1 A Display Containing the Report List

The following code uses the VIEW described in the section "MACRO LOOPING" to build a selection list displaying the REPTID values.

```
INIT:
   dsid = open('parms.runrpts'); /* open rept view */
   call setrow(attrn(dsid,'NLOBS'),1,' ','N');
   call set(dsid);
return;

GETROW:
   rc=fetchobs(dsid,_currow_);
return;

PUTROW:
   submit continue;
     %report(reptid=&reptid,
             title=Report For &reptid)
   endsubmit;
   rc = unselect(_currow_); /* unselect selected option*/
return;

MAIN:
return;

TERM:
   dsid = close(dsid);
return;
```

The advantage of using the parameter file from the section "REPORTING" is that there are fewer data sets to maintain. However, the user will have to select from a list that may not be very descriptive. A parameter file of reports with a report description variable would improve the appearance of the table. The data set in Table 6 can be used as a parameter file to define the selections in an extended table. The variable REPTID specifies the unique report name. REPTDESC specifies the text that will appear on the menu. ORDER specifies the sort order of the items in the list. The data set will have a uniquely defined index on REPTID.

Table 6 Report List Parameter File

REPTID	REPTDESC	ORDER
WIDGET1	The First Widget Report	10
GADTHING	The Gadget Report	20
OTHER	Another Report	30

To present the data set in Table 6 in an extended table, the only change in the SCL code shown previously is the name of the data set passed to the OPEN function. The variable REPTDESC must also be added to the display. Its value can be passed to the report macro as the value of the TITLE parameter.

Through the use of extended tables, we have a mechanism by which we can have a list of reports stored in a data set. By editing that data set, we can maintain the report choices without modifying the source code.

Maintaining the Parameter Files That Drive the System

As we have seen, a data-driven approach to applications development has many advantages, the most important of which is the ability to change the application easily and quickly. To add a new report to the system, add an observation to the data set containing the list of the reports; to add a new line to a report, add observations to the report definition data set. Although changes such as these are not difficult to do on an ad-hoc basis (for example, by using PROC FSEDIT), the file maintenance also can be managed within the application by using data-driven methods.

In the previous section, an extended table was used to display a list of available reports. This technique can also be used to maintain the parameter files. A data set is created, containing a list of all of the data sets that control the system. An extended table is used to display this list of parameter files, allowing users to select the parameter file they wish to edit or modify. When a manager needs to change a subtotal calculation, the PARMS.REPTLINE data set is selected, and the appropriate observations are modified. When the user adds functionality that requires another parameter file, an observation is added to the PARMS.PARMLIST data set, which allows easy access to the file in the future.

Once the user has made a selection from the extended table, the system uses the other information in the PARMS.PARMLIST data set to determine what to do. For many parameter files, calling PROC FSEDIT with a custom screen may be sufficient. Other files may be more complicated and may require calling a special SAS/AF PROGRAM entry, written to handle the editing. Output 3 shows a portion of our PARMS.PARMLIST data set.

Parameter File Data Set Name	AF Module Called to Process File	BY variables for Sort	Name of Format to Create
PARMS.PRODUCTS		PRODUCT	
PARMS.WAREHSE		WHSECODE	
PARMS.REPTLINE		REPTID LINE PRODUCT	
PARMS.LINETEXT		REPTID LINE	
PARMS.MENULIST		ENTRY PROGRAM ORDER	
PARMS.PARMLIST	EDITPARM	ORDER DATASET	

Label Variable for Format	Value Variable for Format	Key Variables for Index	Index Name	Is the Index Unique?	Should I Include Missing
		PRODUCT	PRODUCT	Y	
		WHSECODE	WHSECODE	Y	
		REPTID LINE PRODUCT	_INDEX_	Y	
		REPTID LINE	_INDEX_	Y	
		DATASET	DATASET	Y	

Description Displayed to User	Sort Options	Libref for Format Library	Order as Seen in Displayed List
Valid Products	FORCE		1
Valid Warehouses	FORCE		2
Report Definition Data Set			20
Report Line Text			40
List of Menu Options			990
Parameter Files	FORCE		999

Output 3 Labeled Printout of PARMS.PARMLIST Data Set

The following code fragment from the PUTROW section shows the SCL required to call the editing module.

```
rc=fetchobs(dsid,_currow_);

if module eq " " then
do; /* no program entry specified */

  call fsedit(dataset,'SYSTEM.SCREENS.'||scan(dataset,2,'.'));

end;/* no program entry specified */

else

do; /* call special edit program entry */

  call display(trim(module)||'.PROGRAM',dsid);

end;/* call special edit program entry */
```

There are several common tasks that may need to be performed for a parameter file. For example, parameter files will often need to be sorted, indexed, or used to create formats. To handle these common tasks without needing a special entry for each parameter data set, variables can be added to the PARMS.PARMLIST data set and SCL code added to the PARMLIST entry. In the following code fragment, the SORTVAR variable is used to indicate the variables on which to sort the data set.

```
if sortvars ne "  "  then
do;  /* sort variables are specified*/

   sortdsid = open(dataset,'U');

   if sortdsid gt 0 then
   do; /*data set opened*/

      rc=sort(sortdsid,sortvars,"/"||sortopts);

      if rc ne 0 then
      do; /*sort failed */

          /*code to handle ERROR messages*/

      end;/*sort failed */

      sortdsid=close(sortdsid);

   end; /*data set opened*/

   else

   do; /*open failed */

       /*code to handle ERROR messages*/

   end;/*open failed */

end; /* sort variables are specified*/
```

To create a format from a parameter data set, the user would specify VALUEVAR and LABELVAR variables, and the entry would submit SAS code to create the format. Additional variables are used to specify the format's name and type. The following code fragment shows how the format is created. An SQL view is created and used as a CNTLIN data set to PROC FORMAT.

```
if fmtname ne "  "  then
do;  /* create a format from parm data set*/

   replace fmtlib 'library=&lib';

   submit sql continue;

      create view makefmt as
      select &labelvar as label,
             &valuevar as start,
             '&fmtname' as fmtname,
             '&fmttype' as type
      from &dataset;

   endsubmit;
```

```
      if symgetn('SQLRC') eq 0 then
      do; /*SQL view created*/

         submit continue;

            proc format cntlin=work.makefmt &fmtlib;
            run;

         endsubmit;

         if symgetn('SYSERR') ne 0 then
         do; /*PROC step failed*/

            /*code to handle problems with PROC FORMAT step*/

         end; /*PROC step failed*/

      end; /*SQL view created*/

      else

      do; /*SQL step failed */

         /*code to handle problems with SQL view creation*/

      end; /*SQL step failed*/

   end; /* create a format from parm data set*/
```

The SCL code fragment that follows shows how an index is created for a parameter data set. The user specifies INDEXVAR, the variable list used to create the index, and can specify additional variables to indicate whether each index variable is unique and whether to include missing values. The UNIQUE option is a convenient way to prevent the user from adding duplicate observations.

```
if indexvar ne " "  then
do;  /* create a simple index for parm data set*/

   indxdsid=open(dataset,'V');/*Open data set in UTILITY mode*/

   if indxdsid gt 0 then
   do; /*data set opened*/

      if upcase(unique) eq "Y" then options = "/UNIQUE";
      else options = "/NONUNIQUE";

      if upcase(missings) eq "Y" then options = options||" MISSING";
      else options = options||" NOMISS";

      rc=icreate(indxdsid,iname,indexvar,options);

      if rc ne 0 then
      do; /*index creation failed */

         /*code to handle problems creating index*/

      end; /*index creation failed */

      indxdsid=close(indxdsid);

   end; /*data set opened*/

   else

   do; /*open failed */

        /*code to handla problems opening data set*/

   end; /*open failed */
           /
end; /* create a simple index for parm data set*/
```

Only those files that require sorting, formatting, or indexing have values specified for the corresponding variables.

If the user wants to edit the PARMS.PARMLIST data set, it must be closed first. The following code fragment is SCL code for the EDITPARM program entry, called by the PARMLIST program entry to handle this editing.

```
entry  dsid       8;

length dataset  $ 17;

INIT:
 dataset=dsname(dsid);

 dsid=close(dsid);

 call fsedit(dataset,"SYSTEM.SCREENS."||scan(dataset,2,'.'));

return;
```

Again we have illustrated a data-driven approach, using data sets to drive the entire parameter file maintenance system, to control the editing of the parameter files, and to determine whether to sort a data set and create formats or indexes. In fact, the very data set that is used to control the parameter file maintenance component is itself one of the data sets maintained through the application.

Building Dynamic Menus

We can extend the concept further by developing a generic extended table that displays menu choices. The implementation of such a routine uses the techniques described above. We simply add another parameter file containing the menu choices and control information for each menu the user sees. The menu shown in Display 2 is a sample main menu for our application. We control the contents of this menu by editing a data set.

```
System Main Menu
Command ===>

        Please Tab to Your Selection and Press Enter to Select


                          Data Entry
                        Manage Tables
                          Reporting
                          SAS/ASSIST
                        Exit System
```

Display 2 Sample Main Menu for Application

CONCLUSION

As we have seen, a data-driven approach is an elegant solution that provides flexibility and maintainability. A data-driven mechanism is appropriate to many programming tasks. For those situations where it can be applied, it will be possible for the user to make a variety of functional changes to the system without altering a single line of source code. Of course, any major new functional requirements may necessitate the addition of new macro routines or program entries. Developers who are in the habit of hard-coding extensive program logic should carefully consider the alternative of a data-driven approach as a better solution.

ACKNOWLEDGMENTS

The following staff members of SAS Consulting Services Inc. contributed to the preparation of this paper:

Donald J. Henderson
Martha F. Johnson
Merry G. Rabb
David S. Septoff
Norman Shusterman
Gregory A. Smith
Robina G. Thornton
Phil H. Van Dusen
C. Jessica Yuan

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

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