Paper S1-10

Executing a PROC from a DATA Step

Jason Secosky, SAS Institute Inc., Cary, NC

ABSTRACT

All programs are built by creatively piecing together simpler parts into a larger whole. In SAS®, the SAS macro facility provides an ability to group and piece together analytic blocks. However, writing complex programs using the SAS macro facility can be difficult and cumbersome. An easier way is to combine two new functions, RUN_MACRO and DOSUBL, with DATA step code. RUN_MACRO and DOSUBL enable DATA step code to immediately execute a macro and act on the result, something that was not possible before SAS® 9.2. This paper presents examples of using these two new functions from DATA step to code programs that are easier to read, write, and maintain.

INTRODUCTION

The DATA step is the general purpose programming language of Base SAS. A DATA step has syntax to ease row-at-a-time operations, external file input and output, and general purpose computation. However, it cannot execute a SAS procedure, wait for the PROC to complete, and operate on the results. Some people assume a DATA step can execute a PROC by using CALL EXECUTE. CALL EXECUTE merely queues SAS source to be executed after the DATA step completes. The code is not immediately executed.

In SAS 9.2, we added the ability to execute a PROC from a DATA step. This ability makes writing SAS programs easier by enabling you to perform computations where convenient or necessary, instead of unnecessarily decomposing the program into a series of separately running DATA and PROC steps.

Two new features allow DATA step programs to execute a PROC and wait for it to complete: the RUN_MACRO and DOSUBL functions.

In SAS 9.2 and later, the RUN_MACRO function executes a macro and waits for it to complete. RUN_MACRO can be called only from a user-written function created with the FCMP procedure. Because RUN_MACRO cannot be directly called from a DATA step, you have to write three blocks of code to execute a PROC from a DATA step:

1. Macro to execute a PROC
2. User-written function to execute the macro
3. DATA step to call the user-written function

In SAS 9.3, SAS introduced the DOSUBL function. DOSUBL is an experimental function that executes SAS code directly from a DATA step. Unlike RUN_MACRO, DOSUBL can be called directly from a DATA step without the need for a user-written function.

Both RUN_MACRO and DOSUBL allow a programmer to execute a PROC from a DATA step and wait for the PROC to complete. This enables DATA step programs to act on a PROC's result, eases writing some types of programs, and adds statistical functionality to the DATA step (Christian 2007).

In the first section of this paper, we describe RUN_MACRO and its sister routine RUN_SASFILE. The next section shows DOSUBL and its sister routine DOSUB. The last part of this paper presents three use cases for RUN_MACRO. The first two examples show executing a PROC from a DATA step and acting on the result. The last example demonstrates how to ease programming by executing a PROC from a DATA step.

This paper assumes experience in DATA step programming, writing macros, and creating user-written functions with PROC FCMP. To learn more about these areas, see the Recommended Reading section at the end of this paper. Throughout the paper, the term *user-written function* refers to functions created with PROC FCMP.

RUN_MACRO AND RUN_SASFILE

RUN_MACRO is a function that executes a macro and does not return until the macro completes. RUN_MACRO takes one or more parameters. The first parameter is the name of the macro to execute. Additional parameters are used to create and initialize macro variables that the macro can access. The macro variables created have the same names as the parameters and the macro variable initial values are the values of the parameters. RUN_MACRO returns zero if it was able to execute the macro and nonzero if it could not execute the macro. RUN_MACRO can be called only by a user-written function.

Using RUN_MACRO to execute a PROC from a DATA step requires writing three blocks of code, described in the Introduction: a macro to execute a PROC, a user-written function to execute the macro, and a DATA step to call the user-written function. Here is an example that shows the three blocks of code to create a data set of distinct values with PROC SQL from a DATA step:

Macro to Execute PROC

```
%macro distinct_values;
  %let input_table = %sysfunc(dequote(&input_table));
  %let column = %sysfunc(dequote(&column));
  %let output_table = %sysfunc(dequote(&output_table));

  proc sql;
    create table &output_table as
    select distinct &column
    from &input_table;
%mend;
```

User-Written Function to Execute Macro

```
proc fcmp outlib=sasuser.funcs.sql;
  function get_distinct_values(input_table $, column $, output_table $);
    rc = run_macro('distinct_values', input_table, column, output_table);
    return (rc);
  endsub;
```

DATA Step to Call User-Written Function

```
options cmplib = sasuser.funcs;
data _null_;
  rc = get_distinct_values('sashelp.shoes', 'region', 'work.regions');
run;
```

When this code executes, the DATA step calls GET_DISTINCT_VALUES, which executes %DISTINCT_VALUES, which executes PROC SQL. The SQL query creates a data set named WORK.REGIONS that contains all the unique values for the variable REGION in SASHELP.SHOES. Not only did we get the unique values for REGION, we can use our new user-written function, GET_DISTINCT_VALUES, to get unique values from any data set.

The macro, %DISTINCT_VALUES, removes quotation marks from macro variables, and then uses these macro variables to parameterize a PROC SQL query. In a moment we describe the need for removing quotation marks.

The user-written function GET_DISTINCT_VALUES calls RUN_MACRO to execute the macro %DISTINCT_VALUES. Before executing %DISTINCT_VALUES, RUN_MACRO creates macro variables for its second and higher parameters. In this case, the parameters INPUT_TABLE, COLUMN, and OUTPUT_TABLE cause the macro variables &INPUT_TABLE, &COLUMN, and &OUTPUT_TABLE to be created. The initial values for the macro variables are the values of the parameters.

When macro variables are created for character parameters, the value of the macro variable is placed in double quotation marks. In this example, the INPUT_TABLE parameter contains the value sashelp.shoes. When the macro variable &INPUT_TABLE is created, its value is "sashelp.shoes", where the double quotation marks are part of the macro variable's value. In %DISTINCT_VALUES, the code is invalid with the quotation marks, so they are removed. Numeric parameters are not enclosed in quotation marks when they are placed into macro variables.

Finally, GET_DISTINCT_VALUES is called from a DATA step. When GET_DISTINCT_VALUES is called, it does not return control to the DATA step until %DISTINCT_VALUES completes. Waiting to return until the macro completes enables the DATA step to act on the results of the macro from within the same step. In this case, the result of the macro is a data set. To use the data set from the DATA step that created it, the OPEN and FETCH functions can be used. The SET statement cannot be used because SET opens the data set before the step begins execution and the data set would not have been created yet.

In addition to using data sets to communicate results from a macro, macro variable values can be communicated to the user-written function that calls RUN_MACRO. Just before RUN_MACRO completes, it copies macro variable values back into parameter values. This is done for all macro variables that have a same-named RUN_MACRO parameter. For example, if a parameter to RUN_MACRO is named INPUT_TABLE, when RUN_MACRO completes, the value in macro variable &INPUT_TABLE is copied into INPUT_TABLE. This copy-back mechanism enables you

to place results in a macro variable and "return" the value to the user-written function. Returning values via macro variables can be more convenient than using data sets.

This example shows how to return a comma-separated list of distinct values for a data set:

Macro to Execute PROC

```
%macro distinct_values;
  %let input_table = %sysfunc(dequote(&input_table));
  %let column = %sysfunc(dequote(&column));

  proc sql noprint;
    select distinct &column
    into :csv_distinct_values separated by ','
    from &input_table;
%mend;
```

User-Written Function to Execute Macro

```
proc fcmp outlib=sasuser.funcs.sql;
  function return_distinct_values(input_table $, column $) $;
    length csv_distinct_values $ 32767;
    rc = run_macro('distinct_values', input_table, column, csv_distinct_values);
    return (csv_distinct_values);
  endsub;
```

DATA Step to Call User-Written Function

```
options cmplib=sasuser.funcs;
data _null_;
  vals = return_distinct_values('sashelp.shoes', 'region');
run;
```

In the user-written function RETURN_DISTINCT_VALUES, the macro %DISTINCT_VALUES is executed. In this macro, PROC SQL sets the macro variable &CSV_DISTINCT_VALUES to a comma-separated list of unique values. When the macro completes, the value in &CSV_DISTINCT_VALUES is copied into the parameter to RUN_MACRO with the same name, CSV_DISTINCT_VALUES. Returning macro variable values makes operating on PROC results even simpler than processing a data set or external file. For character values, the only limitation is the size of a variable, 32767 bytes.

RUN_SASFILE is a function that operates just like RUN_MACRO. Instead of taking the name of a macro to execute as its first parameter, RUN_SASFILE takes the name of a fileref. The file referred to by the fileref must contain SAS code that RUN_SASFILE immediately executes. RUN_SASFILE creates and initializes macro variables before executing the SAS code just like RUN_MACRO does. It also copies the values of macro variables back into RUN_SASFILE parameters when the SAS code completes. Examples using RUN_SASFILE can be found in Base SAS documentation for PROC FCMP.

DOSUBL AND DOSUB

DOSUBL is an experimental function in SAS 9.3 that takes a character value and executes the value as SAS code. DOSUBL does not return until the SAS code completes execution. DOSUB is similar to DOSUBL, yet it takes a fileref for a file that contains SAS code. DOSUB reads the file into memory and executes the contents as SAS code.

Both DOSUBL and DOSUB return zero if the code was able to execute and nonzero if the code did not execute. For example, if DOSUB could not read the input file or allocate enough memory to hold the file contents, it returns a nonzero value.

Here is an example of using DOSUBL from a DATA step to create a data set with PROC SQL:

```
data _null_;
  rc = dosubl('proc sql;
               create table work.regions as
               select distinct region from sashelp.shoes;');
run;
```

When DOSUBL is called, it does not return until the SAS code completes. The SAS code passed to DOSUBL uses PROC SQL to create a data set named WORK.REGIONS that contains all the unique values for REGION in SASHELP.SHOES. If you wanted to operate on the distinct values within the same step, you can use the DATA step OPEN and FETCH functions to open and read WORK.REGIONS. The SET statement cannot be used because SET opens the data set before the step begins execution and the data set would not have been created yet.

This example shows using DOSUB instead of DOSUBL to create a data set with the unique values for REGION in SASHELP.SHOES. If we place the PROC SQL code from the prior example in a text file named distinct_regions.sas, here is how DOSUB can execute the SAS code in the file:

```
filename sascode 'distinct_regions.sas';
data _null_;
  rc = dosub('sascode');
run;
```

DOSUBL and DOSUB can communicate results to a DATA step with a data set or an external file. Unlike RUN_MACRO, DOSUBL and DOSUB cannot communicate macro variable values to a DATA step. A future version of SAS might support a mechanism for DOSUBL and DOSUB to communicate macro variable values to a DATA step.

While DOSUBL is appropriate for simple uses, we use RUN_MACRO in the examples in this paper because of its more sophisticated way of communicating values into and out of macro variables.

PROCESSING THE RESULTS OF A PROC

Many analyses in SAS are performed with PROCs. PROCs conveniently bundle together a set of similar analytic algorithms. With RUN_MACRO and DOSUBL, you can access these analytic pieces from a DATA step and act on the result in that DATA step. Being able to execute analytic components from a DATA step extends what the programmer can do in a single step. The next two examples show executing a PROC from a DATA step and acting on the results from the same step.

The first example shows how to execute a PROC SQL query from a DATA step and open and operate on the query result set. The second example executes PROC HTTP from a DATA step to get the driving distance between two addresses from a Web service. Each of the examples uses RUN_MACRO with the "three blocks of code" programming pattern defined in the Introduction: a macro that executes a PROC, a user-written function to execute the macro, then a DATA step to call the user-written function.

EXECUTING AN SQL QUERY FROM A DATA STEP

The DATA step and PROC SQL are used to manage and operate on data. Now that you can execute a PROC from a DATA step, it is possible to execute SQL statements from a DATA step and act on the results. In this example, we create a macro named %SQL_SUBMIT, which executes an SQL statement contained in a macro variable. To execute %SQL_SUBMIT from a DATA step, we create a user-written function named SQL_SUBMIT that calls RUN_MACRO to execute %SQL_SUBMIT. A similar technique is presented by (Poling 2011).

While we need user-written functions to call RUN_MACRO, they also make code modular. Simple functions can be written, tested, and then combined to create more complex functions. In this example, we show this modularity by writing two additional functions, SQL_OPEN and SQL_CLOSE, which call SQL_SUBMIT. SQL_OPEN and SQL_CLOSE mimic the OPEN and CLOSE functions built into Base SAS. Yet instead of opening and closing data sets, like OPEN and CLOSE do, they "open" and "close" the result set of an SQL query.

To open an SQL result set, SQL_OPEN takes an SQL query as its first parameter, creates a temporary SQL view for the query, and opens the view with the OPEN function. In the DATA step, the FETCH function is called to retrieve the results of the query. SQL_CLOSE closes and deletes the view. Both SQL_OPEN and SQL_CLOSE call SQL_SUBMIT to submit SQL statements to perform their action. Here is the code for the example:

Macro to Execute PROC

```
%macro sql_submit;
proc sql;
  %sysfunc(dequote(&sql_code));
%mend;
```

This macro uses PROC SQL to execute the code in &SQL_CODE. The quotation marks RUN_MACRO places around the value in &SQL_CODE are not needed and are removed. There are cases where having the quotation marks is convenient. In this situation, we must remove them for the SQL query to execute correctly.

User-Written Function to Execute Macro

```
proc fcmp outlib=sasuser.funcs.sql;
function sql_submit(sql_code $);
  rc = run_macro('sql_submit', sql_code);
  return (rc ^= 0 or symget('SYSERR') ^= '0');
endsub;

function sql_open(sql_query $);
  length view_name $ 41;
  length sql_code $ 256;

  view_name = cat('tmpvw_', int(ranuni(0)*1E15));
  sql_code = catx(' ', 'create view', view_name, 'as', sql_query);
  rc = sql_submit(sql_code);
  if rc then return (0);

  dsid = open(view_name);
  return (dsid);
endsub;

function sql_close(dsid);
  length view_name $ 41;
  length sql_code $ 256;

  if dsid <= 0 then return(1);

  view_name = attrc(dsid, 'LIB') || '.' || attrc(dsid, 'MEM');
  rc = close(dsid);
  sql_code = catx(' ', 'drop view', view_name);
  rc = sql_submit(sql_code);
  return (rc);
endsub;
run;
```

The SQL_SUBMIT user-written function executes the %SQL_SUBMIT macro and returns 0 if the macro executed successfully. Otherwise, it returns 1.

The SQL_OPEN user-written function uses a random number to create a name for a temporary SQL view. The SQL view encapsulates the submitted SQL query. SQL_OPEN calls SQL_SUBMIT to create the view. If the view was successfully created, the OPEN function is called. A handle to the open data set is returned.

The SQL_CLOSE user-written function fetches the temporary view name from the data set handle, calls the CLOSE function to close the view, and lastly calls SQL_SUBMIT to drop the view.

DATA Step to Call User-Written Function

```
options cmplib = sasuser.funcs;
data _null_;
  length name $ 8 sex $ 1 age weight height 8;
  file print ods=(variables=(name sex age weight height));

  dsid = sql_open('select * from sashelp.class');
  if dsid = 0 then stop;

  call set(dsid);
  rc = fetch(dsid);
  do while (rc = 0);
    put _ods_;
    rc = fetch(dsid);
  end;
  rc = sql_close(dsid);
  stop;
run;
```

This DATA step calls SQL_OPEN to open the result set of an SQL query. SQL_OPEN returns a data set handle that is passed to CALL SET and FETCH. CALL SET causes values to be automatically moved from the result set into DATA step variables when FETCH is called. In this program, we read all the observations and output them in a report. We could have performed other operations on the values retrieved from the SQL query. When the DATA step completes, SQL_CLOSE is called to clean up what SQL_OPEN created.

This example shows how to execute a PROC from a DATA step, and then act on the results. The PROC is PROC SQL and the DATA step dynamically generates a query to execute. The DATA step reads the rows from the query and acts on those rows. In this case, the DATA step merely creates a report of the rows. We also see the beginning of a modular program, with more complex functions, like SQL_OPEN, calling simpler functions like SQL_SUBMIT and OPEN. Modular programming makes debugging, reusing code, and maintenance easier. The next example shows acting on a result returned from a macro variable.

COMPUTE DRIVING DISTANCE WITH PROC HTTP

Several companies provide information about the Internet as a Web service. A Web service accepts an HTTP request to a particular URL and returns a result for the request. SAS provides several ways to use Web services. One of them is PROC HTTP. This example shows a DATA step executing PROC HTTP to retrieve the driving distance between two addresses. This is done by creating a user-written function that executes a macro to fetch the driving distance from the Google Directions API.

The structure of this program is the same as the last example: a macro that executes a PROC, a user-written function to execute the macro, then a DATA step to call the user-written function.

Macro to Execute PROC

```
%macro sas_getDist;
data _null_;
  length url $ 2048;
  url = catt(
    'http://maps.googleapis.com/maps/api/directions/xml?origin=', &start,
    '&destination=', &end,
    '&sensor=false&alternatives=false');
  url = translate(trim(url), '+', ' ');
  call symputx('REQUEST_URL', url);
run;

filename dist temp;
proc http
  out = dist
  url = "%superq(REQUEST_URL)"
  method = "GET"
  ct = "application/x-www-form-urlencoded";
run;

filename  xml_map '<**insert-path**>/google-maps-dist.map';
libname   dist xml xmlmap=xml_map;
data _null_;
  set dist.distance;
  /*  1 mile = 1,609.344 meters */
  dist = dist / 1609.344;
  call symputx('DIST', dist);
run;
%mend;
```

This macro has three parts:

1. **Building the Web service URL**

   The CATT function is used to put together the parts of the URL. The Google API uses plus signs (+) instead of spaces. The TRANSLATE function replaces spaces with plus signs. Macro could have been used to create the URL. However, the author is more familiar with DATA step. To create more maintainable code, a DATA step is used.

   We set the length of the URL variable to 2048 bytes because Google's API limits URLs to a length of 2048 characters.

2. **Making the HTTP request**

   PROC HTTP is used to make the API request. We request an XML response and that response is written to a temporary file, DIST. The URL is passed to %SUPERQ to ensure ampersands (&) in our URL are not treated as the start of a macro variable.

   If your site uses a proxy to access the Internet, you need to use the PROC HTTP PROXYHOST= and PROXYPORT= options to access the Google Directions API.

   This macro could be extended to use PROC HTTP's HEADEROUT= option to store the status of the API request to a file, and then read the file to determine if there was an error during the request. For simplicity, this program does not perform this check. The last example in this paper performs this type of checking.

3. **Retrieving the result from the response**

   We use a DATA step and the XML LIBNAME engine to extract the distance from the API response and convert the distance from meters to miles. The XML map used to extract the distance was created with the SAS XML Mapper and is located in Appendix A. For the XML map in Appendix A to work with this example, save it to a file named google-maps-dist.map and place the path to the file in the FILENAME statement for the XML_MAP fileref. The distance is placed in the macro variable &DIST. In the next section, we see how &DIST is communicated back to the DATA step via the RUN_MACRO parameter named DIST.

User-Written Function to Execute Macro

```
proc fcmp outlib=sasuser.funcs.web;
function sas_getDist(start $, end $);
  rc = run_macro('sas_getDist', start, end, dist);
  return (dist);
endsub;
quit;
```

Our user-written function is named SAS_GETDIST. Placing a prefix on user-written function names, in this case, "SAS_", helps prevent a user-written function name from conflicting with a current or future built-in SAS function name. SAS_GETDIST calls the RUN_MACRO function to execute the macro %SAS_GETDIST.

Macro variables are created for each of the parameters to RUN_MACRO. In this case, macro variables &START, &END, and &DIST are created. The values of the macro variables are initialized to the values of the parameters START, END, and DIST. The values in &START and &END are inserted in the URL used with PROC HTTP.

When %SAS_GETDIST completes, the values in &START, &END, and &DIST are copied into the same-named parameters passed to RUN_MACRO. The values copied into START and END are their original values. However, the value copied into DIST is the result of our PROC HTTP request, the driving distance. This copy-back mechanism enables us to retrieve individual values computed in a macro.

DATA Step to Call User-Written Function

```
data addresses;
  length name $ 32 address $ 128;
  infile datalines truncover;
  input name address $128.;
datalines;
SAS       SAS Campus Dr., Cary, NC 27513
YMCA      1603 Hillsborough St., Raleigh, NC 27605
NCSU      Joyner Visitor Center, NC State University, Raleigh, NC 27695-7504
UNC-CH    UNC Visitor Center, 250 East Franklin Street, Chapel Hill, NC
;

options cmplib=sasuser.funcs;
data distances;
  set addresses;
  dist = sas_getDist('SAS Campus Dr., Cary, NC, 27513', address);
run;

proc print data=addresses;
  where sas_getDist('SAS Campus Dr., Cary, NC, 27513', address) < 10;
run;
```

After creating a data set with sample destinations in it, SAS_GETDIST is called from a DATA step to compute the distance between SAS Headquarters in Cary, NC, and the sample destinations. This example shows how to return a value from a PROC to a DATA step. This is done using the macro variable copy-back mechanism of RUN_MACRO.

A user-written function can be called anywhere a function can be called in SAS, including WHERE clauses, PROC SQL-computed columns, %SYSFUNC, and other user-written functions. This example uses PROC PRINT with a WHERE clause that calls SAS_GETDIST to subset the addresses to those within 10 miles of SAS Headquarters.

To help debug and understand this style of code, it might be useful to turn on the MPRINT, SYMBOLGEN, and MLOGIC system options. These options turn on macro diagnostic output that is useful when understanding the processing that takes place.

Encapsulating the use of the Google API in a user-written function and macro makes it easier for you to change the Web service that is used. If you wanted to use the MapQuest API, all you would need to do is modify the macro. After changing the macro, all callers of SAS_GETDIST would use the MapQuest API instead of the Google API. This ease of maintenance is an attribute that comes with more modular code.

These two examples show how to execute a PROC, or any SAS code, from a DATA step. The first example shows how to process the results of a PROC that creates a data set by using the OPEN and FETCH functions. The second example shows how to return an individual result to a DATA step. All of this is done with the RUN_MACRO function

called from a user-written function. User-written functions can be called wherever a function can be called in SAS, extending the scope of where a PROC can be executed beyond the DATA step.

SIMPLER PROGRAMMING

Executing a PROC from a DATA step can simplify programming. In this example, we create a report from a DATA step and would like to include a generated graph or image in the report for each observation. We use PROC HTTP to retrieve a map image from the Google Maps Image API for each location in the report.

We could retrieve images using a second pass through the data with a macro. However, we are already generating our report with a DATA step, so retrieving the images within our DATA step is more convenient. And, there might be some performance improvement by making one pass through the data instead of two.

This example follows the same three steps as the earlier examples: create a macro that executes a PROC, write a user-written function that executes the macro, and then call the user-written function from a DATA step.

Macro to Execute PROC

```
%macro sas_getMap;
data _null_;
  length url $ 2048;
  url = catt(
    'http://maps.googleapis.com/maps/api/staticmap?markers=', &address,
    '&zoom=12',
    '&size=', &height, 'x', &width,
    '&sensor=false');
  url = transtrn(trim(url), ' ', '%20');
  call symputx('REQUEST_URL', url);
run;

filename img_fref &img_filename;
filename hdr_out temp;
proc http
  out = img_fref
  url = "%superq(REQUEST_URL)"
  headerout = hdr_out
  method = "GET"
  ct="application/x-www-form-urlencoded"
;
run;

data _null_;
  infile hdr_out;
  input;
  if prxmatch('/^HTTP\/\S+\s+\d+/', _infile_);
  status = scan(_infile_, 2, ' ');
  if status = '200' then
    call symputx('STATUS', 0);
  else
    call symputx('STATUS', status);
  stop;
run;
%mend;
```

This macro is similar to the earlier macro used to compute driving distances. This macro uses a DATA step to create a URL for the request to the Google API. The macro variables &ADDRESS, &HEIGHT, and &WIDTH are used to create the URL. Then, PROC HTTP makes the request and stores the response, a PNG image file, in a file. In this example, we use the PROC HTTP HEADEROUT= option to write the response header to a temporary file. At the end of the macro, we retrieve the HTTP status from the header and store it in macro variable &STATUS.

User-Written Function to Execute Macro

```
proc fcmp outlib=sasuser.funcs.web;
function sas_getMap(address $, height, width, img_filename $);
  length status 8;
  rc = run_macro('sas_getMap', address, height, width, img_filename, status);
  return (status);
endsub;
quit;
```

To execute %SAS_GETMAP, we create the user-written function named SAS_GETMAP. As with earlier examples, RUN_MACRO creates macro variables with the same names as the input parameters to RUN_MACRO. In this case, the macro variables &ADDRESS, &HEIGHT, &WIDTH, &IMG_FILENAME, and &STATUS are created. The macro variable initial values are the values of the parameters passed to RUN_MACRO.

When %SAS_GETMAP completes, the values in macro variables are copied back into the same-named parameters to RUN_MACRO. In particular, the value in &STATUS is set by the macro to the HTTP status for the request. The value of &STATUS is copied into the parameter STATUS. STATUS is then returned to the caller of SAS_GETMAP to indicate the success or failure of the Web service request.

DATA Step to Call User-Written Function

```
data addresses;
  length name $ 32 address $ 128;
  infile datalines truncover;
  input name address $128.;
datalines;
SAS       SAS Campus Dr., Cary, NC 27513
YMCA      1603 Hillsborough St., Raleigh, NC 27605
NCSU      Joyner Visitor Center, NC State University, Raleigh, NC 27695-7504
UNC-CH    UNC Visitor Center, 250 East Franklin Street, Chapel Hill, NC
;

options cmplib=sasuser.funcs;
data _null_;
  set addresses;

  /* <Imagine report generation code here> */

  rc = sas_getMap(address, 160, 120, catt('c:\temp\', name, '.png'));
  if rc then
    putlog 'ERROR: Could not get map for ' name;
run;
```

The first step generates input data to use in our report. The second step generates our report. Part of the report creation calls SAS_GETMAP to get a map image for each of our locations. The image is 160x120 pixels, is placed in the directory **`C:\TEMP`**, and has a filename based on the variable NAME in the input data.

We could have retrieved the images using a second pass through the data with a macro. However, we are already generating our report with a DATA step, so retrieving the images within our DATA step is more convenient. And, there might be some performance improvement in making one pass through the data instead of two.

Note, the Google APIs used in this paper are not SAS products. They have their own terms of use that should be reviewed before using.

CONCLUSION

DATA step is the general purpose language of Base SAS. In the past a DATA step could not invoke a PROC and act on the PROCs results from within the same step. In SAS 9.2, the DATA step can execute a PROC via a user-written function that calls the RUN_MACRO function. In SAS 9.3, the experimental function DOSUBL enables executing a PROC from a DATA step without the need for a user-written function. Executing a PROC from a DATA step enables you to write programs that were difficult or cumbersome to write in the past. In this paper we have shown how to use RUN_MACRO and DOSUBL to make SAS coding more modular, simpler, and easier to maintain.

REFERENCES

Christian, Stacey, and Jacques Rioux. 2007. "Adding Statistical Functionality to the DATA Step with PROC FCMP." *Proceedings of SAS Global Forum 2007 Conference*. Cary, NC: SAS Institute Inc. Available at http://support.sas.com/resources/papers/proceedings10/326-2010.pdf.

Poling, Jeremy W. 2011. "Can't Decide Whether to Use a DATA Step or PROC SQL? You Can Have It Both Ways with the SQL Function!" *Proceedings of South Central SAS Users Group 2011 Conference*. Available at http://www.scsug.org/SCSUGProceedings/2011/poling1/SQL%20Function.pdf.

ACKNOWLEDGMENTS

The author would like to acknowledge Stacey Christian and her team for implementing PROC FCMP and the RUN_MACRO function. In addition, Rick Langston implemented DOSUBL and DOSUB and Al Kulik added the ability to call FCMP functions from the DATA step. The author would like to thank Amber Elam and Scott McElroy for performing a technical review of this paper.

RECOMMENDED READING

- Google. *Google Directions API*. Available at http://code.google.com/apis/maps/documentation/directions/.
- Google. *Google Maps Image API*. Available at http://code.google.com/apis/maps/documentation/imageapis/index.html.
- SAS Institute Inc. *The FCMP Procedure*. Available at http://support.sas.com/documentation/cdl/en/proc/63079/HTML/default/viewer.htm#p10b4qouzgi6sqn154ipglazix2q.htm
- SAS Institute Inc. *The HTTP Procedure*. Available at http://support.sas.com/documentation/cdl/en/proc/63079/HTML/default/viewer.htm#n0bdg5vmrpyi7jn1pbgbje2atoov.htm.
- SAS Institute Inc. *SAS Language Reference: Concepts*. Available at http://support.sas.com/documentation/cdl/en/lrcon/62753/HTML/default/viewer.htm#titlepage.htm.
- SAS Institute Inc. *SAS Macro Language: Reference*. Available at http://support.sas.com/documentation/cdl/en/mcrolref/62978/HTML/default/viewer.htm#titlepage.htm.
- SAS Institute Inc. *SAS XML LIBNAME Engine: User's Guide*. Available at http://support.sas.com/documentation/cdl/en/engxml/63177/HTML/default/viewer.htm#titlepage.htm
- Secosky, Jason. "User-Written DATA Step Functions." *Proceedings of SAS Global Forum 2007 Conference.* Available at http://www2.sas.com/proceedings/forum2007/008-2007.pdf.

CONTACT INFORMATION

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

Jason Secosky
SAS Campus Drive
SAS Institute Inc.
E-mail: Jason.Secosky@sas.com

APPENDIX A

The XML LIBNAME engine uses an XML map to create tabular data from an XML file. The XML map in this section extracts the driving distance between two addresses for XML returned by the Google Directions API. This XML map was created with the SAS XML Mapper.

```
<?xml version="1.0" encoding="UTF-8"?>
<SXLEMAP name="google_maps_dist" version="1.2">
  <TABLE name="distance">
    <TABLE-PATH syntax="XPath">/DirectionsResponse/route/leg/distance</TABLE-PATH>
      <COLUMN name="dist">
        <PATH syntax="XPath">/DirectionsResponse/route/leg/distance/value</PATH>
        <TYPE>numeric</TYPE>
        <DATATYPE>integer</DATATYPE>
      </COLUMN>

      <COLUMN name="dist_text">
        <PATH syntax="XPath">/DirectionsResponse/route/leg/distance/text</PATH>
        <TYPE>character</TYPE>
        <DATATYPE>string</DATATYPE>
        <LENGTH>8</LENGTH>
      </COLUMN>
  </TABLE>
</SXLEMAP>
```