SCSUG 2019

Yo Mama is Broke Cause Yo Daddy is Missing: Autonomously and Responsibly Responding to Missing or Invalid SAS® Data Sets Through Exception Handling Routines

Troy Martin Hughes

ABSTRACT

Exception handling routines describe processes that can autonomously, proactively, and consistently identify and respond to threats to software reliability, by dynamically shifting process flow and by often notifying stakeholders of perceived threats or failures. Especially where software (including its resultant data products) supports critical infrastructure, has downstream processes, supports dependent users, or must otherwise be robust to failure, comprehensive exception handling can greatly improve software quality and performance. This text introduces Base SAS® defensive programming techniques that identify when data sets are missing, exclusively locked, or inadequately populated. The use of user-defined return codes as well as the &SYSCC (system current condition) automatic macro variable is demonstrated, facilitating the programmatic identification of warnings and runtime errors. This best practice eliminates the necessity for SAS practitioners to routinely and repeatedly check the SAS log to evaluate software runtime or completion status. Finally, this text demonstrates wrapping exception handling routines within modular, reusable code blocks that improve both software quality and functionality.

YO DADDY

YO.Daddy represents some prerequisite data set that is required by dependent (or downstream) users or processes. Thus, under normal circumstances, the data set exists, is unlocked (for either shared or exclusive access), is formatted appropriately, and contains the expected quantity of data. Note that the directory of the YO library must be initialized by the user.

For example, YO.Daddy can be created with the following DATA step:

```
%let yo=d:\sas\; * USER MUST CHANGE LOCATION;
libname yo "&yo";

data yo.daddy;
   length somevar $20;
   somevar="I'm your daddy!"; output;
   somevar="No, I'm your daddy!"; output;
   somevar="Who's your daddy?!"; output;
run;
```

However, one dark and stormy night, the data set is mistakenly missing, which can be simulated with the following DELETE procedure:

```
proc delete data=yo.daddy;
run;
```

With YO.Daddy missing, dependent processes will fail—either through functional failures that produce warnings or runtime errors, or worse yet, through processes that create invalid or erroneous data sets or data products. Software reliability demands that quality assurance measures be put into place that can detect the exception—the missing data set—to prevent these failures, and *exception handling* describes the process of programmatically identifying software threats and responding accordingly in real-time. SAS exception handling is introduced and thoroughly demonstrated in the following papers:

- *Why Aren't Exception Handling Routines Routine? Toward Reliably Robust Code through Increased Quality Standards in Base SAS®* (Hughes, 2015)
- *Ushering SAS® Emergency Medicine into the 21st Century: Toward Exception Handling Objectives, Actions, Outcomes, and Comms* (Hughes, 2019)

YO MAMA

YO.Mama represents a data set that relies on YO.Daddy—that is, when YO.Daddy is missing, invalid, corrupt, or does not contain sufficient data, YO.Mama will either be missing or invalid because dependent processes that create or transform YO.Mama will fail.

For example, under normal circumstances, the following DATA step and SORT procedure both creates and transforms YO.Mama:

```
data yo.mama;
   set yo.daddy;
run;

proc sort data=yo.mama;
   by somevar;
run;
```

However, with YO.Daddy deleted or otherwise missing, the following output demonstrates two runtime errors and one warning:

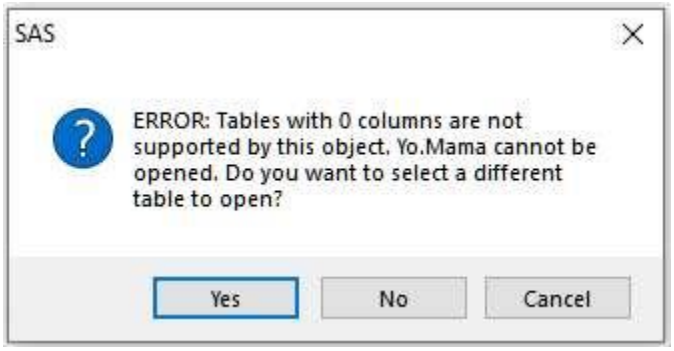
```
data yo.mama;
   set yo.daddy;
ERROR: File YO.DADDY.DATA does not exist.
run;

NOTE: The SAS System stopped processing this step because of errors.
WARNING: The data set YO.MAMA may be incomplete.  When this step was stopped there
were 0
         observations and 0 variables.
NOTE: DATA statement used (Total process time):
      real time           0.00 seconds
      cpu time            0.00 seconds


proc sort data=yo.mama;
   by somevar;
ERROR: Variable SOMEVAR not found.
run;

NOTE: The SAS System stopped processing this step because of errors.
NOTE: PROCEDURE SORT used (Total process time):
      real time           0.00 seconds
      cpu time            0.01 seconds
```

Oddly, despite the first runtime error (i.e., the missing data set), the YO.Mama data set is nevertheless created—albeit with zero variables and zero observations. This useless, empty data set cannot be opened, and double-clicking on its icon (from within SAS Display Manager or SAS Enterprise Guide) produces the following popup:



EXCEPTION HANDLING – MISSING DATA SET

Basic exception handling can be implemented that will first test for the existence of YO.Daddy (utilizing the EXIST function) before referencing this data set in the subsequent DATA step:

```
%macro test;
%if %sysfunc(exist(yo.daddy)) %then %do;
   data yo.mama;
      set your_daddy;
   run;
   %end;
%else %put ERROR: YO.Daddy is missing!;
%mend;

%test;


ERROR: YO.Daddy is missing!;
```

This form of exception handling is sometimes described and differentiated as exception *reporting* because it *reports* the exception (in this case, to the log). Utilizing "ERROR:" in the %PUT statement causes the subsequent message to be printed in red (by SAS default) to the log. More importantly, however, the programmatic detection of the missing data set prevents the DATA step from executing, thus preventing the runtime error that would have resulted.

Although exception reporting to the SAS log can be useful during software testing and debugging, the SAS log is generally useless in production environments that run automated processes that must be reliably executed. Thus, when software demands higher reliability, exceptions should be detected programmatically, in which case the SAS automatic macro values &SYSCC and &SYSERR are invaluable tools. This case is made more substantially in the author's text: *Pinching Off Your SAS® Log: Adapting from Loquacious to Laconic Logs To Facilitate Near-Real Time Log Parsing, Performance Analysis, and Dynamic, Data-Driven Design and Optimization* (Hughes, 2018).

For example, the following NO_DADDY macro now sets the &SYSCC (i.e., system current condition) automatic macro variable to 2 if YO.Daddy is missing:

```
%macro no_daddy();
%let syscc=0;
%if %sysfunc(exist(yo.daddy)) %then %do;
   data yo.mama;
      set yo.daddy;
   run;
   %end;
%else %do;
   %put ERROR: YO.Daddy is missing!;
   %let syscc=2;
   %end;
%mend;

%no_daddy;
%put SYSCC: &syscc;


ERROR: YO.Daddy is missing!;
SYSCC: 2
```

The output demonstrates that the missing data set was detected, the subsequent DATA step was avoided, and &SYSCC was changed to 2 to reflect the failure. Subsequent processes can test the value of &SYSCC to assess whether or not they should execute. Note that other runtime errors encountered by the SAS system can also modify the value of &SYSCC, and &SYSCC *generally* will retain only the *highest* error code.

For example, the following CREATE_AND_SORT macro now tests for the existence of YO.Daddy before both the subsequent DATA step and SORT procedure:

```
%macro create_and_sort();
%let syscc=0;
%if %sysfunc(exist(yo.daddy)) %then %do;
   data yo.mama;
      set yo.daddy;
   run;
   %end;
%else %do;
   %put ERROR: YO.Daddy is missing!;
   %let syscc=2;
   %abort;
   %end;
%if %sysfunc(exist(yo.mama)) %then %do;
   proc sort data=yo.mama;
      by somevar;
   run;
   %end;
%mend;

%create_and_sort;
%put SYSCC: &syscc;


ERROR: YO.Daddy is missing!;
SYSCC: 20000
```

The %ABORT statement terminates the SAS macro if YO.Daddy is missing, so the second %IF statement is not executed (or even read by SAS) if YO.Daddy is missing. Moreover, %ABORT resets &SYSCC to 20000, so although &SYSCC is briefly set to 2 (if YO.Daddy is missing), it is immediately changed to 20000 thereafter. These idiosyncrasies are important to understand, because even the following SAS code (in which &SYSCC is briefly changed to 99999) will ultimately set &SYSCC to 20000—NOT 99999:

```
%macro create_and_sort();
%let syscc=0;
%if %sysfunc(exist(yo.daddy)) %then %do;
   data yo.mama;
      set yo.daddy;
   run;
   %end;
%else %do;
   %put ERROR: YO.Daddy is missing!;
   %let syscc=99999;
   %abort;
   %end;
%if %sysfunc(exist(yo.mama)) %then %do;
   proc sort data=yo.mama;
      by somevar;
   run;
   %end;
%mend;

%create_and_sort;
%put SYSCC: &syscc;


ERROR: YO.Daddy is missing!;
ERROR: Execution terminated by an %ABORT statement.
SYSCC: 20000
```

This limitation can be overcome by creating a user-defined return code—a global macro variable created inside a macro that can be evaluated programmatically by (parent) processes calling the macro or by subsequent (dependent) processes that require the macro's successful execution. In the following updated macro, the &CREATE_AND_SORTRC return code is declared as a global macro variable and is not overwritten or influenced by the value of &SYSCC:

```
%macro create_and_sort();
%let syscc=0;
%global create_and_sortRC;
%let create_and_sortRC=0;
%if %sysfunc(exist(yo.daddy)) %then %do;
   data yo.mama;
      set yo.daddy;
   run;
   %end;
%else %do;
   %put ERROR: YO.Daddy is missing!;
   %let create_and_sortRC=2;
   %abort;
   %end;
%if %sysfunc(exist(yo.mama)) %then %do;
   proc sort data=yo.mama;
      by somevar;
   run;
   %end;
%mend;

%create_and_sort;
%put SYSCC: &syscc;
%put RC: &create_and_sortRC;


ERROR: YO.Daddy is missing!;
ERROR: Execution terminated by an %ABORT statement.
SYSCC: 20000
RC: 2
```

An alternative solution replaces the %ABORT statement with %RETURN, in which case the value of &SYSCC is no longer modified by the SAS system:

```
%macro create_and_sort();
%let syscc=0;
%global create_and_sortRC;
%let create_and_sortRC=0;
%if %sysfunc(exist(yo.daddy)) %then %do;
   data yo.mama;
      set yo.daddy;
   run;
   %end;
%else %do;
   %put ERROR: YO.Daddy is missing!;
   %let create_and_sortRC=2;
   %return;
   %end;
%if %sysfunc(exist(yo.mama)) %then %do;
   proc sort data=yo.mama;
      by somevar;
   run;
   %end;
%mend;

%create_and_sort;
%put SYSCC: &syscc;
```

```
%put RC: &create_and_sortRC;


ERROR: YO.Daddy is missing!;
SYSCC: 0
RC: 2
```

The %RETURN function only terminates the currently executing macro, whereas %ABORT terminates the entire program that is executing. With YO.Daddy again created, the following code now executes without exception or other failure:

```
data yo.daddy;
   length somevar $20;
   somevar="I'm your daddy!"; output;
   somevar="No, I'm your daddy!"; output;
   somevar="Who's your daddy?!"; output;
run;

%create_and_sort;
%put SYSCC: &syscc;
%put RC: &create_and_sortRC;


SYSCC: 0
RC: 0
```

EXCEPTION HANDLING – EMPTY DATA SET

YO.Daddy is now in the picture, but is he useful? Not necessarily, especially if he is gutless (i.e., has no data). For example, the following code recreates YO.Daddy with neither variables nor observations, an exception not previously tested by CREATE_AND_SORT:

```
data yo.daddy;
run;

%create_and_sort;
%put SYSCC: &syscc;
%put RC: &create_and_sortRC;
```

The code produces a runtime error (because of an unhandled exception), as shown in the SAS log:

```
%create_and_sort;

NOTE: There were 1 observations read from the data set YO.DADDY.
NOTE: The data set YO.MAMA has 1 observations and 0 variables.
NOTE: DATA statement used (Total process time):
      real time           0.00 seconds
      cpu time            0.00 seconds


ERROR: Variable SOMEVAR not found.

NOTE: The SAS System stopped processing this step because of errors.
NOTE: PROCEDURE SORT used (Total process time):
      real time           0.00 seconds
      cpu time            0.00 seconds

%put SYSCC: &syscc;
SYSCC: 3000
%put RC: &create_and_sortRC;
RC: 0
```

To overcome this limitation, a macro can be constructed that will evaluate not only the existence of a data set but also the number of observations contained therein. By parameterizing the data set name (as the DSN parameter), the CHECKSOMEDATA macro is more configurable and reusable and thus improves software quality:

```
* RC set to 1 if data set exists, 0 if it does not;
%macro checksomedata(dsn= /* data set name */);
%global checksomedataRC;
%let checksomedataRC=%sysfunc(exist(&dsn));
%mend;

%checksomedata(dsn=oh_mommy);
%put RC: &checksomedataRC;


RC: 0
```

In this example, the Oh_mommy data set does not exist, so &CHECKSOMEDATARC is set to 0, the return value of the EXIST function when the referenced data set is missing. Note the disparity in that a &SYSCC value of 0 represents no warning or runtime error has occurred, whereas some functions (such as EXIST) produce a 0 return value when some desirable condition (like data set existence) is unmet.

The design can be modified to not only *generate* but also *return* the return code, in a modular design pattern often referred to as a *user-defined SAS macro function*. User-defined macro functions are generally more flexible (than macro non-functions) because they can be referenced inline with other SAS statements; however, they cannot contain step boundaries such as SAS procedures or DATA steps (because they can often be invoked from *inside* SAS procedures or DATA steps). For example, CHECKSOMEDATA has now been modified to directly return the return code produced by the EXIST function:

```
* RV set to 1 if data set exists, 0 if it does not;
%macro checksomedata(dsn= /* data set name */);
%sysfunc(exist(&dsn))
%mend;

* this data set exists;
%put RV: %checksomedata(dsn=yo.mama);

* this data set does not exist;
%put RV: %checksomedata(dsn=oh_mommy);
```

The first invocation returns 1 because YO.Mama exists, whereas the second invocation returns 0 because Oh_mommy does not:

```
* this data set exists;
%put RV: %checksomedata(dsn=yo.mama);
RV: 1

* this data set does not exist;
%put RV: %checksomedata(dsn=oh_mommy);
RV: 0
```

Also note that the %SYSFUNC line does not terminate with the usual semicolon; this is necessary because a value is being returned from a user-defined macro function.

EXCEPTION HANDLING – EXCLUSIVELY LOCKED DATA SET

In other cases, a data set may exist but may be exclusively locked by some process so the data set cannot be modified or otherwise accessed. Data set existence can be evaluated with the OPEN function, which will produce a return value of 0 when it cannot obtain an exclusive lock, thus indicating that some other user or process has an exclusive lock on the data set. *Note that an exclusive (aka read-write) lock is required to create, modify, or otherwise write to a data set.*

The CHECKSOMEDATA macro can be modified to include the ability to evaluate file lock status:

```
* return value can have the following values;
* 0 - DSN missing;
* 1 - DSN exists and is not locked;
* 2 - DSN exists and is locked;
%macro checksomedata(dsn= /* data set name in LIB.DSN or DSN format */);
%local rc lib dsid close;
%if %index(&dsn,.)=0 %then %let lib=WORK;
%else %do;
   %let lib=%scan(&dsn,1,.);
   %let dsn=%scan(&dsn,2,.);
   %end;
%* test for DSN existence;
%if %sysfunc(exist(&lib..&dsn))=1 %then %do;
   %let rc=1;
   %* test for DSN exclusive lock;
   %let dsid=%sysfunc(open(&lib..&dsn));
   %if &dsid>0 %then %do;
      %let close=%sysfunc(close(&dsid));
      %end;
   %else %let rc=2;
   %end;
%else %let rc=0;
&rc
%mend;
```

Note that as additional exceptions are evaluated, additional return code values are often required to describe the diversity of issues that could arise. With this complexity is endowed the requirement that program documentation adequately describes return code values so they can be utilized by developers. Although not discussed extensively in this text, SAS file locking (and the inherent weaknesses therein) is dissected in painful detail in the following papers:

- *From a One-Horse to a One-Stoplight Town: A Base SAS® Solution to Preventing Data Access Collisions through the Detection and Deployment of Shared and Exclusive File Locks* (Hughes, 2015)
- *A Waze App for Base SAS®: Automatically Routing Around Locked Data Sets, Bottleneck Processes, and Other Traffic Congestion on the Data Superhighway* (Hughes, 2016)
- *Stress Testing and Supplanting the SAS® LOCK Statement: Implementing Mutex Semaphores To Provide Reliable File Locking in Multiuser Environments To Enable and Synchronize Parallel Processing* (Hughes, 2017)

EXCEPTION HANDLING – INADEQUATELY SIZED DATA SET

Size matters, and in some cases, if YO.Daddy is too small, he should be ignored. Previous examples have demonstrated how a zero-variable-zero-observation data set can be inexplicably created, so one of the most basic quality control tests is to ensure that a data set has at least one observation—thus identifying the exception of an empty data set. In other cases, higher observation-count thresholds should be set for data set size, and can be parameterized into SAS macro procedures or functions.

First, recreate YO.Daddy with his original three observations:

```
data yo.daddy;
   length somevar $20;
   somevar="I'm your daddy!"; output;
   somevar="No, I'm your daddy!"; output;
   somevar="Who's your daddy?!"; output;
run;
```

CHECKSOMEDATA can be modified one final time, allowing it to detect and handle not only zero-observation data sets but also data sets that have fewer observations than some user-specified threshold:

```
* return value can have the following values;
* 0 - DSN missing;
* 1 - DSN exists and is not locked;
* 2 - DSN exists and is locked;
* 3 - DSN has too few observations;
%macro checksomedata(dsn= /* data set name in LIB.DSN or DSN format */,
   minobs= /* minimum number of observations that must exist */);
%local rc lib dsid close nobs;
%if %index(&dsn,.)=0 %then %let lib=WORK;
%else %do;
   %let lib=%scan(&dsn,1,.);
   %let dsn=%scan(&dsn,2,.);
   %end;
%* test for DSN existence;
%if %sysfunc(exist(&lib..&dsn))=1 %then %do;
   %let rc=1;
   %* test for DSN exclusive lock;
   %let dsid=%sysfunc(open(&lib..&dsn));
   %if &dsid>0 %then %do;
      %let nobs=%sysfunc(attrn(&dsid,nobs));
      %if &nobs<=&minobs %then %let rc=3;
      %let close=%sysfunc(close(&dsid));
      %end;
   %else %let rc=2;
   %end;
%else %let rc=0;
&rc
%mend;
```

If you are expecting YO.Daddy to have only one or more observations, then set MINOBS to 1, and the return code (1) will demonstrate that he's of sufficient size (i.e., 3 observations):

```
%put %checksomedata(dsn=yo.daddy, minobs=1);

RC: 1
```

However, if you are expecting YO.Daddy to be a bit larger—say, at least 5 observations—then setting MINOBS to 5 will create a return code (3) demonstrating he's insufficient:

```
%put %checksomedata(dsn=yo.daddy, minobs=5);

RC: 3
```

Evaluating observation count is a common endeavor in any programming language, but is best operationalized through modular, user-defined procedures or functions, such as illustrated in Quentin McMullen's epic paper that showcases the CHECKRECORDCOUNTS macro (McMullen, 2018).

TOWARD SAS USER-DEFINED MACRO FUNCTION REUSABILITY

Commonly implemented functionality (such as testing data set existence, availability, or size) often can be placed inside user-defined functions to be reused in perpetuity. This design and functionality increases the quality of the software being developed as well as the productivity of the SAS practitioner or team developing it. Returning to the original example (in which YO.Mama is derived from YO.Daddy and subsequently sorted), software reuse could be implemented by invoking CHECKSOMEDATA to ensure that both YO.Daddy and YO.Mama exist.

To recap, the baseline functionality (without necessary exception handling) follows:

```
data yo.mama;
   set yo.daddy;
run;
```

```
proc sort data=yo.mama;
   by somevar;
run;
```

The CHECKSOMEDATA macro can now be invoked twice—once to determine if YO.Daddy exists, and subsequently to ensure that YO.Mama exists before she is sorted:

```
%macro isyodaddygoodenough();
%global isyodaddygoodenoughRC;
%let isyodaddygoodenoughRC=;
%if "%checksomedata(dsn=yo.daddy, minobs=1)"="1" %then %do;
   data yo.mama;
      set yo.daddy;
   run;
   %if "%checksomedata(dsn=yo.mama, minobs=1)"="1" %then %do;
      proc sort data=yo.mama;
         by somevar;
      run;
      %end;
   %else %let isyodaddygoodenoughRC=YO Mama failed you!;
   %end;
%else %let isyodaddygoodenoughRC=YO Daddy failed you!;
%mend;


%isyodaddygoodenough;
%put IsYoDaddyGoodEnoughRC: &isyodaddygoodenoughRC;


IsYoDaddyGoodEnoughRC:
```

The return code is blank, indicating that neither YO.Mama nor YO.Daddy failed you. However, after deleting YO.Daddy and invoking the ISYODADDYGOODENOUGH macro a second time, the exception (i.e., the missing data set) is identified and handled:

```
proc delete data=yo.daddy;
run;

%isyodaddygoodenough;
%put IsYoDaddyGoodEnoughRC: &isyodaddygoodenoughRC;


IsYoDaddyGoodEnoughRC: YO Daddy failed you!
```

Importantly, the absence of data sets (or the data within) no longer produces warnings or runtime errors, as SAS software now executes more responsibly and reliably. Additional exception handling could be implemented to detect unrelated exceptions, such as missing variables, extra variables, or variables having the wrong data type. The possibilities are endless and can substantially improve software quality.

CONCLUSION

Sometimes YO.Daddy is there for you, but sometimes he isn't. Sometimes YO.Daddy is big enough, but not always. And sometimes YO.Daddy is accessible, but other times he's locked in the basement with a "Do not disturb" sign on the door. But rather than being surprised by the shortcomings of YO.Daddy, why not treat YO.Mama right and prevent these failures before they occur?! This paper demonstrated exception handling techniques that detect missing, locked, or inadequately sized data sets, eliminating unnecessary warnings and runtime errors that can cripple and corrupt production SAS software and their resultant data products. Moreover, the placement of exception handling methods inside modular, reusable, user-defined SAS macro functions is a best practice that can improve not only the quality of software but also the productivity of the SAS practitioners striving to develop, test, and maintain that software.

REFERENCES

Hughes, T. M. (2015). From a One-Horse to a One-Stoplight Town: A Base SAS® Solution to Preventing Data Access Collisions through the Detection and Deployment of Shared and Exclusive File Locks. *SAS Global Forum.* Dallas, TX. Retrieved from https://support.sas.com/resources/papers/proceedings15/3380-2015.pdf

Hughes, T. M. (2015). Why Aren't Exception Handling Routines Routine? Toward Reliably Robust Code through Increased Quality Standards in Base SAS®. *SAS Global Forum (SGF).* Dallas, TX. Retrieved from https://support.sas.com/resources/papers/proceedings15/3387-2015.pdf

Hughes, T. M. (2016). • A Waze App for Base SAS®: Automatically Routing Around Locked Data Sets, Bottleneck Processes, and Other Traffic Congestion on the Data Superhighway. *SAS Global Forum (SGF).* Las Vegas, NV. Retrieved from https://www.lexjansen.com/wuss/2016/96_Final_Paper_PDF.pdf

Hughes, T. M. (2017). • Stress Testing and Supplanting the SAS® LOCK Statement: Implementing Mutex Semaphores To Provide Reliable File Locking in Multiuser Environments To Enable and Synchronize Parallel Processing. *SAS Global Forum (SGF).* Orlando, FL. Retrieved from https://support.sas.com/resources/papers/proceedings17/1465-2017.pdf

Hughes, T. M. (2018). Pinching Off Your SAS® Log: Adapting from Loquacious to Laconic Logs To Facilitate Near-Real Time Log Parsing, Performance Analysis, and Dynamic, Data-Driven Design and Optimization. *PharmaSUG.* Seattle, WA. Retrieved from https://www.pharmasug.org/proceedings/2018/AD/PharmaSUG-2018-AD07.pdf

Hughes, T. M. (2019). Ushering SAS® Emergency Medicine into the 21st Century: Toward Exception Handling Objectives, Actions, Outcomes, and Comms. Philadelphia, PA: PharmaSUG. Retrieved from https://www.lexjansen.com/pharmasug/2019/AP/PharmaSUG-2019-AP-071.pdf

McMullen, Q. (2018). Runtime Validation of SAS® Jobs: A Threesome of Error-Throwing Macros. *SAS Global Forum (2018).* Denver, CO. Retrieved from https://www.sas.com/content/dam/SAS/support/en/sas-global-forum-proceedings/2018/1793-2018.pdf

CONTACT INFORMATION

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

Name: Troy Martin Hughes
E-mail: troymartinhughes@gmail.com

SAS and all other SAS Institute Inc. product or service names are registered trademarks or trademarks of SAS Institute Inc. in the USA and other countries. ® indicates USA registration.

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