Paper CC-19

Don't Get the Finger... You Know the FAT Finger
Creating a Modular Report Approach using SAS® Macros

Penny Eckert, Equifax Inc.

ABSTRACT

As SAS report developers we often use the same bits and pieces of code for different reports and analyses. We take a morsel of code from this program and combine it with a scrap of code from that program to create a new analysis. While retreading code is something we all do, surprisingly it can be time consuming and can often lead to the dreaded phenomena of fat fingering. By understanding how your organization uses its code base, you can choose the **SAS Macro** techniques to create a modular reporting approach from which you can choose bits and pieces of your most useful code lines, yet minimize the amount of necessary tweaking. By developing this modular approach, you can save time, standardize your report output, minimize coding errors and avoid getting the fat finger.

INTRODUCTION

As SAS professionals we are accustomed to working under deadlines to produce reports or analyses. In order to complete our tasks as quickly as possible, we retread previous programs and code snippets to complete the assignments. Rather than implementing macros or developing a modular reporting approach which would require additional testing and debugging – therefore taking more time - we complete the analysis and then move on to the next task. The irony is that often by trying to work quickly, we don't always work efficiently and, we introduce errors to our work and as a result, add time to the task.

A modular reporting approach is focused on reusing existing code and programs (something we all do) without manually modifying the code (something we all should do, but don't). The process creates a code base or library of programs that are designed to be used for current and future development with little or no code modifications. Macros enable you to reuse code from different programs without modifying the code itself. Modifications or changes that do need to be made to programs can be minimized or can be limited by their use. Adopting a few or all of the following techniques drives a modular reporting approach.

This paper will review three macro techniques: macro variables, macros and autocall libraries. They are the foundation to a modular reporting approach. There are additional macro techniques that aid report development and data processing. For a complete review of the macro language and how code is processed through the macro processor please review ***SAS 9.3 Macro Language: Reference.*** Understanding how the code is compiled and executed will facilitate development. .

AN OUNCE OF ORGANIZATION IS WORTH A POUND OF AGGRAVATION

Surprisingly, two keys to a successful deployment of a modular reporting approach aren't SAS functions and procedures (but you will utilize them and they are *super* cool), but are planning and organization. Consider these three questions:

1. Does your organization consistently reuse the same programs or code lines in some shape or form? If, so which programs? What code lines?
2. Do multiple people use a version of the same program?
3. What portions of the programs are modified with each execution? Do you change the datasets involved in processing? Do you filter the data or report?

The answers to these questions will shape the **SAS MACRO** techniques utilized. For example, if you consistently cut and paste the same snippets of code into different programs, then using **%INCLUDE** or setting up a macro library may solve your needs. If you only need to modify a few dates or data sources to execute your programs, using **%LET** to create a macro variable to flow through your programs may a better approach. By posing these questions, you will provide insight on the best approach to take to develop a modular reporting approach.

DO YOU CHANGE TITLES, DATES OR DATASETS REFERENCES?

If you change titles, dates or dataset references within a report, adding macro variables to your programs may alleviate some basic code changes. **%LET** statements may limit the need make the routine code changes. **%LET** defines a macro variable and assigns a value to it. A macro variable enables a user to declare the value of a variable ***one*** time and then that value will flow throughout the program. Macro variables created through **%LET** are efficient

tools to push title changes, dataset changes, and date changes in a program. The standard syntax of a **%LET** statement is:

```
%LET macro-variable-name=value;
```

The assigned value must be a string. However, unlike standard variables, macro variables have no size limitations. The macro variable is referenced or used in the program by prefixing '&' to the variable name. The below **%LET** statement creates a macro variable region. This limits the code modifications to the **%LET** statement and provides the flexibility to change the macro variable in future executions. This seemingly small **MACRO** function can reap large rewards in the battle against the fat finger.

```
%LET region=Europe;

title "Average MSRP by Car Make";
title2 "&region"; -becomes -> where origin="Europe" during execution.
PROC sql;
    select make, avg(msrp)
    from sashelp.cars
    where origin="&region" -becomes -> where origin="Europe" during execution.
    group by make;
quit;
```

%LET is also a good tool to use set librefs in the libname statement. This can be handy when source directory paths are lengthy. Below is a Unix directory path.

```
%LET dir=/my_primary_directory/my_project_directory/data_directory;

libname abc "&dir";
```

If the source directory changes, modifications are limited to the **%LET** statement. If your organization utilizes a consistent directory structure, a single **%LET** macro variable reference can be used to reference multiple directories.

```
%LET dir=/my_primary_directory/my_project_directory;

libname abc "&dir/DATA";
libname def "&dir/RPTS"; -> needs double quotes to be read by macro processor
```

There are other subtleties or 'gotchas', around macro variables and how they are referenced and processed. When a macro variable is used within quotes, double quotes are necessary. If a macro variable is used in combination with a letter, underscore, number or period, the macro variable would be referenced with a with a period. The period tells the macro processor to identify the macro variable from the rest of the word or code.

```
%LET lib=sashelp;
 set &lib..cars; -> extra '.' tells SAS where macro variable ends
```

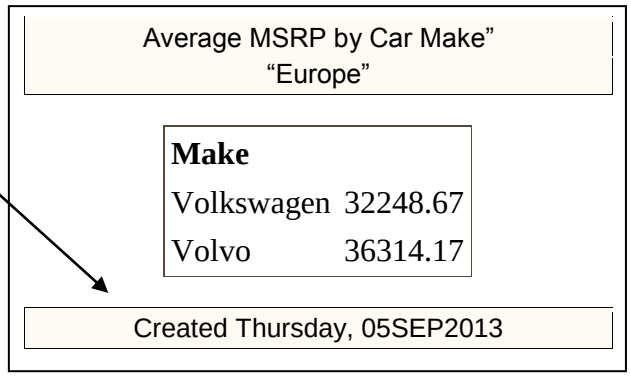
%LET creates ***user*** defined macro variables. There are also ***system or automatic*** macro variables. As the name implies, these macro variables provide information about the SAS system environment. These include information about the operating system, current job ids and the most recently created SAS dataset. For a complete listing of all system macro variables review ***SAS® 9.3 Macro Language: Reference: Macro Variables Defined by the Macro Processor.*** The uses of these macro variables may not be apparent at first glance. The listing in the following table demonstrates some possible applications of these automatic provided macro variables.

SYSTEM MACRO VARIABLE	DESCRIPTION	POSSIBLE APPLICATION
SYSDATE/SYSDATE9	Date at SAS invocation or SAS session start	Add dates to report titles, foot notes. Use in data processing.
SYSDAY/SYSTIME	Provides day of the week/ or time at SAS invocation or SAS session start	Add dates to report titles, foot notes. Use in data processing.

SYSTEM MACRO VARIABLE	DESCRIPTION	POSSIBLE APPLICATION
SYSLAST	Provides name of the most recently created SAS data set.	Verify that dataset is created before executing report.
SYSERR	Provides a return code set by SAS procedures and the DATA step	Check the return code provided by the macro and abort the job if there is an error.

Table 1. Selected System Macro Variables

```
%LET region=Europe;
title "Average MSRP by Car Make";
title2 "&region";
footnote "Created &sysday, &sysdate9";
PROC sql;
    select make, avg(msrp)
    from sashelp.cars
    where origin="&region" and make
like "V%"
    group by make;
quit;
```



Output 1. Output with System Macro Variable

%PUT YOUR MACRO WHERE YOUR LOG IS

If reducing errors is one of the goals of using macro variables, then it's important to check that the values being passed to the program are correct. The **%PUT** statement writes the values of the macro variables in the SAS log so that they can be verified.

```
%put region;
```

```
%put &region;
Asia
```

Specific macro variables can be listed or ***all*** macro variables, ***all automatic***, or ***all user*** defined macro variables can be listed in the log.

```
%put region;
%put _all_;
%put _automatic_;
%put _user_;
```

```
%put _automatic_;
AUTOMATIC AFDSID 0
AUTOMATIC AFDSNAME
AUTOMATIC AFLIB
AUTOMATIC AFSTR1
AUTOMATIC FSPBDV
AUTOMATIC SYSADDBITS 64
AUTOMATIC SYSBUFFR
AUTOMATIC SYSCC 3000
AUTOMATIC SYSCHARWIDTH 1
AUTOMATIC SYSCMD
AUTOMATIC SYSDATE 05SEP13
AUTOMATIC SYSDATE9 05SEP2013
AUTOMATIC SYSDAY Thursday
----------
```

Output 2. Log Output from %Put _automatic_

The listings provided by the **%PUT** function can be lengthy. It is helpful to specify the macro variables to check to make the log easier to read.

A WORD ABOUT DEBUGGING

In addition to **%PUT**, there are also a series of system options which add notes to the log when a macro or macro variable is executed that provide information to debug programs. A combination of **MPRINT**, **SYMBOLGEN,** and/ or **MLOGIC** will provide details of a macro or macro variables execution, or if there is a problem lack of execution. Below is a macro and the log provided using these options.

```
option symbolgen mprint mlogic;

%MACRO avg_make(country);
title "Average MSRP by Car Make";
PROC SQL;
select make, avg(msrp)
from sashelp.cars
where origin="&country" and make
like "V%"
group by make;
quit;
%MEND;

%avg_make(Europe)
```

```
%avg_make(Europe)
MLOGIC(AVG_MAKE):  Beginning execution.
MLOGIC(AVG_MAKE):  Parameter COUNTRY has value
Europe
MPRINT(AVG_MAKE):   title "Average MSRP by Car
Make";
MPRINT(AVG_MAKE):   PROC SQL;
SYMBOLGEN:  Macro variable COUNTRY resolves to
Europe
MPRINT(AVG_MAKE):   select make, avg(msrp) from
sashelp.cars where origin="Europe" and make like
"V%" group by make;
MPRINT(AVG_MAKE):   quit;

MLOGIC(AVG_MAKE):  Ending execution.
```

Output 3. Log Output Using Debugging Options

Table 2 provides the key system options that can be used for macro implementation and debugging.

SYSTEM OPTION	DESCRIPTION	SAS LOG NOTES EXAMPLE
MPRINT	Notes are added to the SAS log that detail the text produced from a macro.	`MPRINT(AVG_MAKE):  title "Average MSRP by Car Make";` `MPRINT(AVG_MAKE):  proc sql;`
SYMBOLGEN	Notes are added to the SAS log that details how macro variables are resolved within the macro.	`SYMBOLGEN:  Macro variable COUNTRY resolves to Europe`
MLOGIC	Notes are added to the SAS log that describes the execution of the macro.	`MLOGIC(AVG_MAKE): Beginning execution.` `MLOGIC(AVG_MAKE): Parameter COUNTRY has value Europe`
SERROR	Adds notes to the log when an ampersand and word combination is found, but the macro variable cannot be resolved.	`WARNING: Apparent symbolic reference GROUPBY not resolved.`
MERROR	Adds notes to the log when an % and word combination is found, but the macro cannot be resolved.	`WARNING: Apparent invocation of macro AVG_REGION not resolved.`

Table 2. SAS Macro Debugging System Options

When the development is complete the options can be removed or the option can be set to off by using the NO version of the option. Like some of the **%PUT** functions, they very useful during development, but the options increase the log output and can be quite cumbersome. It's recommended that the options be turned off when the solution is in production.

DOES YOUR ORGANIZATION CONSISTENTLY REUSE THE SAME PROGRAMS OR CODE LINES IN SOME SHAPE OR FORM?

In many organizations, the same program or code lines are used for many different analyses. Why reinvent the wheel? A small change here and a small change there, and maybe a piece of new code here and the analysis is complete. If your organization uses the same programs or code snippets, transforming the code into macros that can be called or used without modifying the main code base, may help eliminate error and speed development. The first task is to identify those programs and code lines that are consistently utilized. The second task is to transform them to macros.

Macros embody the flexibility of the modular reporting approach. Macros package those code snippets or programs for repeated use. Write once and use as many times as needed, in one program or in many programs. The general syntax of a macro is:

```
%MACRO macro-name;
    macro-text;
%MEND;
```

To use or invoke that macro, the syntax is:

```
%macro-name
```

The ***%MACRO*** signals the start of the macro and the ***%MEND*** signals the end. Macros can range from simple, similar to macro variables, to highly complex. Like macro variables, macros have qualities that are important in regards of a modular reporting approach. First, macros are stored in files. And macros can have parameters, which are simply macro variables that are referenced in the macro definition. By storing the macro is a file, the code can be used for repeated use within and the parameters provide the flexibility to change the content with each execution.

To be effective, macros must be founded on error free code. When writing macro based programs it is best to first write and test the SAS program without any macro content using hard coded constraints and a fixed data set. If you are transforming existing code, this task is complete. When the code is error free ***and*** provides the desired results, add the macro content. By developing in this two stage approach, SAS code syntax and logic errors are isolated from macro syntax and logic errors enabling a less painful development process.

The below macro, ***avg_make*** provides the average MSRP for each automobile make in the dataset cars. To invoke or use the macro, execute ***%avg_make***. Notice that that macro doesn't need a semi-colon as traditional SAS statements do.

```
%MACRO avg_make;
title "Average MSRP by Car Make";
PROC SQL;
select make, avg(msrp)
    label 'Avg MSRP' format dollar10.2
from sashelp.cars
where make like "M%"
orderby make
quit;
%MEND;

%avg_make
```

Average MSRP by Car Make"

Make	Avg MSRP
MINI	$18,499.00
Mazda	$21,770.73
Mercedes-Benz	$60,656.81
Mercury	$27,972.78
Mitsubishi	$23,423.62

Output 4. Macro %avg_make

This macro is useful if the only information needed is about car makes starting with the letter 'M'. What if you need to know the average MSRP of car makes starting with the letter 'V'? From different regions? Parameters increase a macros utility by increasing its flexibility. One macro can provide different results based on the parameter provided. Parameters are macro variables referenced in the macro text. A macro can have any number of parameters.

Parameters can be either keyword or positional and are provided in the **%MACRO** statement and are enclosed in parentheses and separated by commas. Keyword parameters are defined keyword parameters are defined by the name of the parameter and do not have to be included for the macro to execute. Positional parameters are defined only by their order in the macro invocation and must always be included in the macro execution. A macro can contain both positional and keyword parameters, but the positional parameters must come first. The parameters themselves can be null values, text, macro variables and macro references.

```
%MACRO avg_make(letter) ;

title "Average MSRP by Car Make";
PROC SQL;
select make, avg(msrp)
label 'Avg MSRP' format dollar10.2
from sashelp.cars
where make like "&letter.%"
group by make;
quit;
%MEND;
%avg_make(S)
```

use period to define end of macro name

Average MSRP by Car Make"

Make	Avg MSRP
Saab	$37,640.00
Saturn	$17,234.38
Scion	$13,565.00
Subaru	$25,501.82
Suzuki	$16,230.25

Output 5. Macro %avg_make with Parameter

The macro to the left below uses keyword parameters, while the macro to the right uses positional parameter.

```
%MACRO avg_msrp(region=, type=);
title "Average MSRP of &Type";
title2 "&Region";
PROC SQL;
select make, avg(msrp) as Avg_MSRP
from sashelp.cars
where origin="&region" and  type="&type"
group by make;
quit;
%MEND;

%avg_msrp(region=Europe,type=Sedan)
```

```
%MACRO avg_msrp(type, region);
title "Average MSRP of &Type";
title2 "&Region";
PROC SQL;
select make, avg(msrp) as Avg_MSRP
from sashelp.cars
where origin="&region" and  type="&type"
group by make;
quit;
%MEND;
%avg_msrp(Sedan,Europe)
```

Both macros produce the same output.

SNIPPETS TO MACROS

Macros do not need to be complex. Do you or your team members repeat the same functions or code lines in the same program? Identifying these code snippets and making them portable also drives a modular reporting approach. Consider this scenario. A commonly used dataset provides a date in a month-year format. However, all other datasets have complete dates in a mm/dd/yyyy format. The month-year date needs to be recast to the standard mm/dd/yyyy format for consistency and to be used in any calculations.

The code to convert the month-year date into a new date with a default day of '01' is:

```
data test;
mon_date='02/1965';
new_date=mdy(substr(mon_date,1,2),'01',substr(mon_date,4));
format new_date mmddyy10.;
run;
```

What if there numerous columns that need to be recast? Transform the code to a macro.

```
%MACRO get_mm01yyyy(date);
new_&date=mdy(substr(&date,1,2),'01',substr(&date,4));
format new_&date mmddyy10.;
%MEND;

data test;
mon_date='02/1965';
%get_mm01yyyy(mon_date);
run;
```

It is easy to see how transforming those routine data manipulations and code lines can speed development. Previous code is utilized and the number of code lines is minimized. A win-win in the drive to a modular reporting approach.

LOCAL VS GLOBAL

One can see how useful macros are in a modular reporting approach. The same code can be referenced without modifying the core code and it can be used multiple times. Multiple macros can be used within the same report and the same macro can be used within different reports. When using multiple macros it's important to recognize the difference between LOCAL and GLOBAL macro variables. Local macro variables are those defined in inside a macro and can be used only within that macro. Global macro variables are defined outside macros and can be used both outside and within other macros.

In the example below ***&groupby*** is a GLOBAL macro variable while ***&region*** is a LOCAL macro variable because it is defined within the avg_cars macro.

```
%LET groupby=type;
%MACRO avg_cars(region);
title "Average MSRP by &groupby";
title2 "&region";
PROC SQL;
   select &groupby, avg(msrp) as avg_msrp
from sashelp.cars where origin="&region"
group by &groupby;
quit;
%MEND;

%avg_cars(Europe)
```

The macro variable REGION was created within the ***avg_car*** macro. When the macro variable is used with in the macro ***avg_invoice*** an error is noted. This differentiation between LOCAL and GLOBAL becomes important as multiple macros are referenced. For a macro value to flow within multiple macros, which is a common scenario within a modular reporting approach, the macro variable must be GLOBAL

```
%LET groupby=type;

%MACRO avg_cars(region);
title "Average MSRP by &groupby";
title2 "&region";
PROC SQL;
select &groupby, avg(msrp) as avg_msrp
from sashelp.cars
where origin="&region"
group by &groupby;
quit;
%MEND;
%avg_cars(Europe)

%MACRO avg_invoice;
PROC SQL;
select &groupby, avg(msrp) as avg_msrp
from sashelp.cars
where origin="&region"
group by &groupby;
quit;
%MEND;
%avg_invoice
```

```
%MACRO avg_invoice;
56          PROC SQL;
57          select &groupby, avg(msrp) as avg_msrp
58          from sashelp.cars
59          where origin="&region"
60          group by &groupby;
61          quit;
62          %MEND;
63          %avg_invoice
MLOGIC(AVG_INVOICE):  Beginning execution.
MPRINT(AVG_INVOICE):   PROC SQL;
SYMBOLGEN:  Macro variable GROUPBY resolves to type
WARNING: Apparent symbolic reference REGION not
resolved.
SYMBOLGEN:  Macro variable GROUPBY resolves to type
MPRINT(AVG_INVOICE):   select type, avg(msrp) as
avg_msrp from sashelp.cars where origin="&region"
group by type;
NOTE: No rows were selected.
MPRINT(AVG_INVOICE):   quit;
NOTE: PROCEDURE SQL used (Total process time):
      real time           0.00 seconds
      cpu time            0.00 seconds
MLOGIC(AVG_INVOICE):  Ending execution.
```

Output 6. Log avg_invoice Macro

A macro variable can be declared as global by using the syntax:

```
%global region;
```

Using the debugging options helps understand if and how your macro variables and macros are resolved. When in doubt, use **%PUT** to list GLOBAL or LOCAL (%put _local_) macro variables in the log.

```
%put _global_;
%put _local_;
```

DO MULTIPLE PEOPLE USE A VERSION OF THE SAME PROGRAM?

It is very common for organizations to use multiple versions of the same program. Version control is a challenge in many organizations. Programs are handed down from analyst to analyst, and along the way they are morphed and changed. Data sources change, business rules change and for good or bad, developers have the inherent desire to place their stamp on the code they use. The result is often code that is changed beyond the original intent. Having a single source of code for multiple users solves this problem. It has additional benefits: it frees programmers to concentrate on new development, provides one version of the 'truth', and facilitates training new team members.

The answer to this problem is to create a central repository that contains the programs or macros for all users to access and utilize. **%INCLUDE** and **SAS Autocall** libraries provide the two main ways to access programs and or macros located in an centralized external location. The first step to implement either technique is to designate a directory or directories to maintain the macros and programs. User access should be limited, so that the macros or programs can be edited by a select few. This helps maintain version control and if all users use the same core macros and programs for analysis contributes to creating one version of the truth.

%INCLUDE

%INCLUDE enables the execution an external file or program from another program. Contents of the referenced program can range from full programs with multiple procedures and data manipulation, mere lines of code, and macros. The general **%INCLUDE** syntax is:

```
%include '/specified-program-directory/program-name.sas /source;
```

A filename reference can also be used. The fileref references the program which can later be used with the **%INCLUDE .**

```
Filename carspgm '/specified-program-directory/program-name.sas';

%INCLUDE carspgm;
```

The source option captures the execution notes of the referenced program in the SAS log. Programs referenced using **%INCLUDE** are immediately executed in the program. You can reference macros and pass macro variables to the programs accessed through the **%INCLUDE**.

Numerous programs can be nested within each other – but there is the possibility that you can have too much of a good thing. If the goal is to reduce the possibility errors, then it's important to maintain traceability of the programs – the more nesting the harder it is to follow your program.

AUTOCALL LIBRARY

A **SAS Autocall** libraries provides functionality similar to **%INCLUDE**. An autocall library is a central location that contains macros as programs for multiple users to access. There are subtle differences on how a SAS autocall library is referenced based on the operating system. Please refer to ***Saving Macros in an Autocall Library*** to verify the specifications of a specific platform. Examples provided in the following examples are based on a Unix platform.

The system option **MAUTOSOURCE** turns on the autocall facility and the **SASAUTOS** option directs SAS to check the listed directory or directories for macros. In the example below, the options direct SAS to checks the directory, 'mydirectory/mac_lib' for macros. Multiple directories can be referenced in the statement.

```
options MAUTOSOURCE sasautos=('/mydirectory/mac_lib,sasautos)  mautolocdisplay;
```

The macros must be saved with the same program names as the macro. So, the macro ***avg_make*** would be saved in a program names *avg_make.sas.* To execute any macro located within the macro library, it is referenced simply by its name, such as within a program.

```
&avg_make
```

%INCLUDE VS AUTOCALL LIBRARY

For the most part, there is little difference to the user between using **%INLCUDE** or using an **AUTOCALL** library. But

it is important to understand the differences between the two techniques. The autocall library uses the name of the macro file while **%INCLUDE** requires the physical name and path of the program.

```
&avg_make -> Autocall library
%include '/specified-program-directory/avg_make.sas /source; -> %INCLUDE
```

Also, a macro being referenced using an autocall library will not be compiled until it is used the first time and thereafter will not be compiled during that current SAS session. A program using a macro reference using **%INCLUDE** will be compiled upon each execution.

A mix of both techniques can drive the best results. **%INCLUDE** can access both traditional programs and macros, while an autocall library requires macro programs. For users not comfortable or just becoming familiar with macros **%INCLUDE** is a straightforward way that can be leveraged to standardize and centralize a code base without necessarily using macro themselves. The brevity that the macro naming convention provides may be more useful when a macro is used as a replacement for code lines or functions.

CONCLUSION

The benefits of adopting a modular reporting approach outweigh the extra time implementing requires. The first step is reviewing the reporting and development code used within your organization. Then based on the needs of your organization, the appropriate macro techniques can be used. Organizations that currently don't employ macros can start by adding macro variables and transforming existing code lines and programs into macros. Organizations that already use macros can focus on creating a centralized library and using %INCLUDE and or autocall libraries so that all members of the organization can access the same code repository and leverage the modular code.

These macro techniques are just a few of all the macro functions available in SAS. However, the techniques reviewed provide the most bang for the effort when developing a modular report approach. Using one or all of these macro functions will help your organization prompt faster development, standardize report output, minimize coding errors and avoid getting the fat finger.

REFERENCES

Delwiche, Lori D. and Susan J. Slaughter, *The Little SAS® Book: A Primer, Second Edition*, Copyright @ 1998, pp. 153-167, Cary, North Carolina: The SAS Institute, Inc.

Aster, Rick, *Professional SAS® Programming Logic*, Copyright @ 2000, pp. 431-446, Paoli, Pennsylvania: Breakfast Communications Corporation

Paper CC-019 *SAS® Macro Autocall and %Include***,** Jie Huang, Merck & Co., Inc. Tracy Lin, Merck & Co., Inc.

ACKNOWLEDGMENTS

I owe much gratitude to Dessa Overstreet for persuading me to write a paper and the support of Equifax management in the writing of the paper itself.

CONTACT INFORMATION

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

Name: Penny Eckert
Enterprise: Equifax, Inc.
E-mail: penny.eckert@equifax.com