Paper 115-2022

SAS® Log Parsing: SAS Logs Without the Slog

Modifying SAS® Log Content and Extracting Important Information

Drew Metz, NORC at the University of Chicago

ABSTRACT

As any SAS programmer has undoubtedly noticed, the size of the SAS Log can grow exponentially with the complexity of the program. This is even more true and unpredictable when your SAS program is driven by an external source – such as when an Excel file is read in line by line to control the operations of a program.

Picking out useful information from the Log can become burdensome when thousands of lines are generated. Further, there are potentially important issues SAS will flag as NOTES within the Log that can be lost among hundreds of innocuous notes. Programming a SAS Log parser can pay off when the SAS Log is growing too large to manually review. Additionally, SAS has several options and system level Macro variables related to the Log that are useful for interacting with the Log in a meaningful way.

There are two key reasons why SAS Log parsing is an approachable task. First, a small bit of SAS code can output the Log as an external text file (ideal for parsing). Second, the patterns of SAS Logs are largely predictable. We know that they adhere to conventions such as errors will always be flagged at the start of the Log line with “ERROR:”. A basic understanding of what messages are output to the Log can be utilized to write a program that reports important information from the Log.

INTRODUCTION

This paper will cover three broad topics related to SAS Logs: System options to control how Errors, Warnings and Notes are output to the Log, Automatic Macro variables related to the Log, and basic methods for extracting information from an external Log file.

The reader will gain the ability to tailor the Log created by their programs to their specific needs. There are SAS options that can be used to control what program execution issues are categorized as Errors, Warnings or Notes; additionally, there are SAS System-level Macro variables which can hold the latest warnings and errors written to the SAS Log. Utilizing these pre-existing tools is a first step to getting more out of the SAS Log.

Using the basic Log parsing methods described in this paper, the reader will be able to extract key information from their SAS Log without needing to manually read through the text.

The author assumes readers have a basic familiarity with macro variables, macro functions as well as the contents of the SAS Log. Throughout this paper exist a series of SAS code examples. Within these sections, comments annotating the code are labeled in green text and surrounded by SAS comment markings like so:

```
/* Example SAS code comment */
```

SAS SYSTEM OPTIONS: MODIFYING WHAT IS WRITTEN TO THE LOG

This section of the paper introduces three SAS System options that control what is output to the Log file and what is categorized as an Error or Warning.

MSGLEVEL is a System Option that can be modified to output additional notes and messages to the SAS Log. By setting MSGLEVEL=I, the SAS Log will contain much more information detailing the execution of SAS procedures. The full list of which can be seen in the primary SAS documentation.

The following code uses the MERGE procedure to combine two SASHELP datasets which share several variables and uses the MSGLEVEL Option to output INFO statements detailing which variables are overwritten:

```
OPTIONS MSGLEVEL=i;
PROC SORT DATA=sashelp.class;
    BY Name;
RUN;
PROC SORT DATA=sashelp.classfit;
    BY Name;
RUN;

DATA Class_Merge_Overwrite;
    MERGE sashelp.classfit sashelp.class;
    BY Name;
RUN;
```

Figure 1. SAS Log displaying INFO statements generated by MSGLEVEL=I option

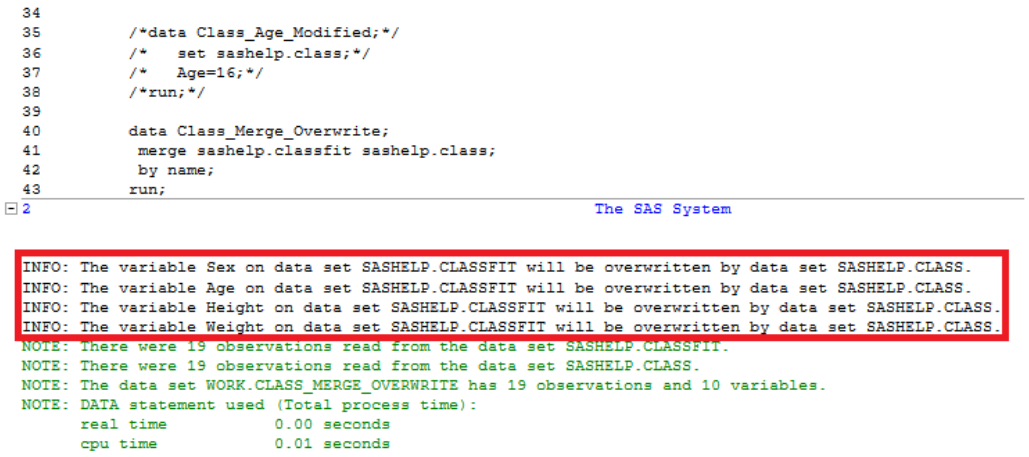


Figure 1. SAS Log displaying INFO statements generated by MSGLEVEL=I option

DKRICOND and DKROCOND are two System Options that can be set to produce an ERROR or a WARNING in the log when a variable specified on a DROP, KEEP, or RENAME statement do not exist. DKRICOND is used for flagging variables in an INPUT dataset and DKROCOND is used for an OUTPUT data set.

By default, DKRICOND is set to ERROR and DKROCOND is set to WARNING. The following code demonstrates setting the DKRICOND and DKROCOND options such that renaming missing variables on the Input data set produces a WARNING and doing so on the Output data set produces an ERROR:

```
OPTIONS DKRICOND=Warning;
DATA Class_Input_Warning;
    /*Missing_Variable does not exist on the Input dataset*/
    SET sashelp.class(RENAME=(Missing_Variable=New_Name));
RUN;

OPTIONS DKROCOND=ERROR;
DATA Class_Output_Error;
    SET sashelp.class;
    /*Last_Name variable does not exist on Output dataset*/
    RENAME Last_Name=Lname;
```

```
RUN;
```

Figure 2. SAS Log displaying WARNING / ERROR statements generated by modifying DKRICOND and DKROCOND options

```
27         options dkricond=Warning;
28         data Class_Input_Warning;
29          *Missing_Variable does not exist on the Input dataset;
30          set sashelp.class(rename=(Missing_Variable=New_Name));
WARNING: Variable Missing_Variable is not on file SASHELP.CLASS.
WARNING: The variable Missing_Variable in the DROP, KEEP, or RENAME list has never been referenced.
31         run;

NOTE: There were 19 observations read from the data set SASHELP.CLASS.
NOTE: The data set WORK.CLASS_INPUT_WARNING has 19 observations and 5 variables.
NOTE: DATA statement used (Total process time):
      real time           0.01 seconds
      cpu time            0.00 seconds


32
33         options dkrocond=ERROR;
34         data Class_Output_Error;
35          set sashelp.class;
36          *Last_Name variable does not exist on Output dataset;
37          rename Last_Name=Lname;
38         run;

ERROR: The variable Last_Name in the DROP, KEEP, or RENAME list has never been referenced.
NOTE: The SAS System stopped processing this step because of errors.
WARNING: The data set WORK.CLASS_OUTPUT_ERROR may be incomplete.  When this step was stopped there were 0 observations and 5
         variables.
```

Figure 2. SAS Log displaying WARNING / ERROR statements generated by modifying DKRICOND and DKROCOND options

Each of these System Options helps the user control what data processing issues are output to the Log. Maintaining control over the scope of what program issues are reported to the Log, and what they are categorized as, can be a preliminary step to SAS Log parsing.

AUTOMATIC MACRO VARIABLES: ACCESS AND UTILIZE ERROR AND WARNINGS

This section of the paper describes automatic Macro variables that hold the most recent Warning and Error messages. In addition, a simple strategy for making use of these Macro variables for process decision making and reporting is described.

During the execution of a program, SAS automatically populates and updates several Macro variables. SYSWARNINGTEXT and SYSERRORTEXT hold the latest WARNING and ERROR messages, respectively.

Additionally, SYSERROR is an automatic Macro variable that holds a coded value to signify what type of ERROR, WARNING or other program execution issue has occurred most recently. Each of these Macro variables can be utilized in a data step to conditionally control a program based on the latest WARNINGS and ERRORS produced.

While the automatic Macro variable can hold a variety of values depending on how a SAS process executes, SYSERROR holds the default value of 0 if no ERROR or WARNING has occurred during SAS processing. Strategic use of SYSERR allows SAS code to be written that is executed conditionally only if ERRORS or WARNINGS were (or were not) encountered. SYSERR can be utilized for conditional processing like so:

```
%IF &SYSERR. NE 0 %THEN %DO;
    /* This code will only be run if no ERROR/WARNINGS were encountered in
    SAS processing */
%END;
```

SAS LOG PARSING: READING THE LOG FILE AND PRODUCING A TAILORED REPORT

The first step in SAS Log parsing is to obtain the Log in a format that is easy to work with. Users of SAS Enterprise Guide or SAS Studio may be familiar with reading Logs directly within these applications.

Using the PROC PRINTTO procedure, SAS will route the Log information to an external file which, when saved in text format, is prime for programmatic consumption. It is important to note that PROC PRINTTO can be understood as a Toggle On/Toggle Off procedure. The first PROC PRINTTO statement declares the location of the external SAS Log file. Until the PROC PRINTTO procedure is called again, all code executed will produce Log information that is output to the specified external file. An example of this syntax is provided below:

```
FILENAME mylog "C:\USER\SAS_LOGS\Example_Log.txt";
PROC PRINTTO LOG=mylog new;
RUN;

/* Code between the two PROC PRINTTO statements will produce log
information saved in Example_Log.txt */

PROC PRINTTO;

/* This code will NOT produce log information saved in Example_Log.txt */
```

Once an external Log file has been saved, SAS can iterate through the text of the Log line by line and extract information that is relevant to the user. The INFILE and INPUT statements provide a way to read in the Log file and process each line, respectively. The most basic parsing of a SAS Log is demonstrated below:

```
DATA Log_File;
    INFILE "C:\USER\SAS_LOGS\Example_Log.txt" TRUNCOVER;
    INPUT Log_Line $1024.;
    /* This code will simply save the content of each line in the SAS Log
    to the variable Log_Line in the dataset Log_File */
RUN;
```

The above example is just a starting point, producing a data set with every line from the Log file stored in the variable Log_Line. The next step is to add logic to the SAS code that is applied to each line of the Log file that is read. This content can be tailored to the specific needs of any SAS programmer.

For each iteration of the above DATA step, SAS processes a single line of the Log file. The Log_Line variable stores a given line as a string, meaning that any string processing functions available in SAS can be utilized to determine if the line is of interest. One of the most basic checks that can be performed on the SAS Log is checking the category of a given Log line. Using the SUBSTR function it can be determined if a line in the Log is a NOTE, INFO, WARNING or ERROR.

As an example, the below code reads each line in a Log file and outputs lines that fall into these four categories:

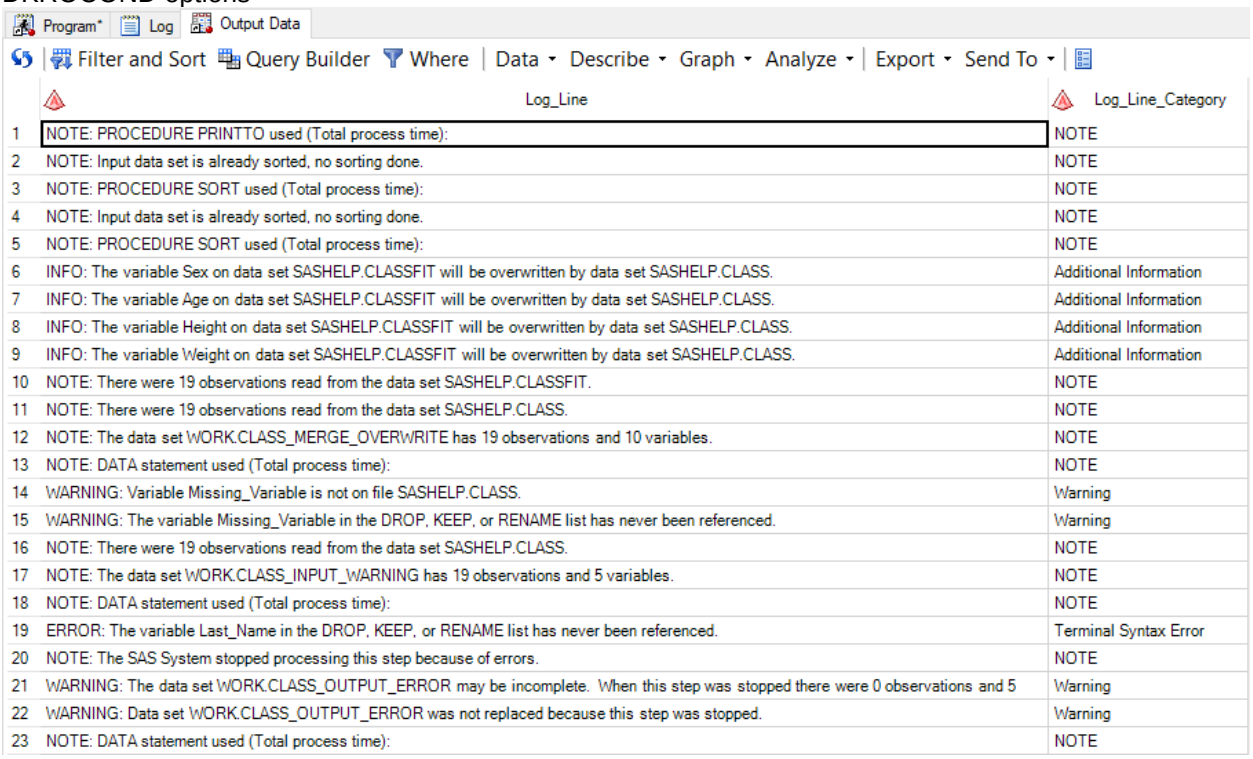
```
DATA Log_File_Report;
    INFILE "C:\USER\SAS_LOGS\Example_Log.txt" TRUNCOVER;
    INPUT Log_Line $1024.;
    /* This code examines the characters at the start of each line of the
    Log file and, if these are founds to be INFO:, NOTE:, ERROR:, or
    WARNING: will output the log line with the relevant category */
    LENGTH Log_Line_Category $24.;

    IF (SUBSTR(Log_Line),1,5)="INFO:") THEN DO;
          Log_Line_Category = "Additional Information";
```

```
        Output;
    END;
    ELSE IF (SUBSTR(Log_Line),1,5)=”NOTE:”) THEN DO;
        Log_Line_Category = “Note”;
        Output;
    END;
    ELSE IF (SUBSTR(Log_Line),1,6)=”ERROR:”) THEN DO;
        Log_Line_Category = “Terminal Syntax Error”;
        Output;
    END;
    ELSE IF (SUBSTR(Log_Line),1,8)=”WARNING:”) THEN DO;
        Log_Line_Category = “Warning”;
        Output;
    END;
RUN;
```

Running the above code example on an external SAS Log file that contains the Log messages highlighted in Figure 1 and Figure 2 produces the following data set summary.

Figure 3. SAS Log displaying WARNING / ERROR statements generated by modifying DKRICOND and DKROCOND options



	Log_Line	Log_Line_Category
1	NOTE: PROCEDURE PRINTTO used (Total process time):	NOTE
2	NOTE: Input data set is already sorted, no sorting done.	NOTE
3	NOTE: PROCEDURE SORT used (Total process time):	NOTE
4	NOTE: Input data set is already sorted, no sorting done.	NOTE
5	NOTE: PROCEDURE SORT used (Total process time):	NOTE
6	INFO: The variable Sex on data set SASHELP.CLASSFIT will be overwritten by data set SASHELP.CLASS.	Additional Information
7	INFO: The variable Age on data set SASHELP.CLASSFIT will be overwritten by data set SASHELP.CLASS.	Additional Information
8	INFO: The variable Height on data set SASHELP.CLASSFIT will be overwritten by data set SASHELP.CLASS.	Additional Information
9	INFO: The variable Weight on data set SASHELP.CLASSFIT will be overwritten by data set SASHELP.CLASS.	Additional Information
10	NOTE: There were 19 observations read from the data set SASHELP.CLASSFIT.	NOTE
11	NOTE: There were 19 observations read from the data set SASHELP.CLASS.	NOTE
12	NOTE: The data set WORK.CLASS_MERGE_OVERWRITE has 19 observations and 10 variables.	NOTE
13	NOTE: DATA statement used (Total process time):	NOTE
14	WARNING: Variable Missing_Variable is not on file SASHELP.CLASS.	Warning
15	WARNING: The variable Missing_Variable in the DROP, KEEP, or RENAME list has never been referenced.	Warning
16	NOTE: There were 19 observations read from the data set SASHELP.CLASS.	NOTE
17	NOTE: The data set WORK.CLASS_INPUT_WARNING has 19 observations and 5 variables.	NOTE
18	NOTE: DATA statement used (Total process time):	NOTE
19	ERROR: The variable Last_Name in the DROP, KEEP, or RENAME list has never been referenced.	Terminal Syntax Error
20	NOTE: The SAS System stopped processing this step because of errors.	NOTE
21	WARNING: The data set WORK.CLASS_OUTPUT_ERROR may be incomplete. When this step was stopped there were 0 observations and 5	Warning
22	WARNING: Data set WORK.CLASS_OUTPUT_ERROR was not replaced because this step was stopped.	Warning
23	NOTE: DATA statement used (Total process time):	NOTE

Figure 3. SAS Log displaying WARNING / ERROR statements generated by modifying DKRICOND and DKROCOND options

In the above data set there are numerous NOTE Log messages reported which may not be of interest. However, removing all NOTE messages from the Log summary is not optimal, as specific NOTE messages can contain useful information. As an example, when a Data step references a variable that is not on the input data set or otherwise created, a NOTE message is generated that reports that the variable is uninitialized. This Log message takes the form “NOTE: Variable [variable name] is uninitialized”.

Using the knowledge of how SAS structures specific Log messages, a SAS Log parser can be designed to narrow the scope of what content is reported from the Log.

The above section of code that uses SUBSTR to determine if a line in the Log is a NOTE message can be modified to identify the specific uninitialized variable NOTE:

```
ELSE IF (SUBSTR(Log_Line),1,5)="NOTE:") THEN DO;
      IF (FIND(Log_Line,'is uninitialized') > 0) THEN DO;
            Log_Line_Category = "Uninitialized Variable";
            OUTPUT;
      END;
END;
```

This modified code makes use of the FIND function to see if the text "is uninitialized" exists within a line of the Log that has already been identified as a NOTE message.

The FIND function searches the text of the first argument and returns the starting character position of the second text argument within the first argument string. If the second argument is not found within the first argument, FIND returns a value of 0. Therefore, if FIND returns a value greater than 0 the first argument contains the second argument.

The result of using this check is that only NOTES referencing uninitialized variables will be output to the Log report, rather than every NOTE encountered in the Log.

Introducing nested IF statements and specific targeting of known SAS Log messages allows a SAS Log parsing program to do the work of sorting through a SAS Log for information considered important.

CONCLUSION

The SAS Log is often the first thing a SAS user views after the execution of a program to determine if there were any issues. The more complex a SAS program is the less sufficient a passing glance at the SAS Log becomes.

Using the tools SAS provides, such as SAS System Options and SAS Automatic Macro variables, the user has some degree of control over what is reported to a SAS Log and how a program can directly interact with ERROR and WARNING messages.

Further, the exporting of the SAS Log to an external text file transforms the Log as something the user manually reviews to something that can be consumed by a SAS program. As demonstrated within this paper, basic SAS string functions such as SUBSTR and FIND are all that is necessary to determine if a given line of the SAS Log should be output to a summary or report of the Log.

As a final exercise, the author invites the reader to consider how more complex SAS programs can make use of a SAS Log parser. For instance, consider the example of a SAS program that is primarily driven by a Macro function that performs a set of actions upon a provided data set. An abstract view of this program may have this appearance:

```
%macro Do_Operation(dset=);
      FILENAME mylog "C:\USER\SAS_LOGS\Example_Log.txt";
      PROC PRINTTO LOG=mylog new;
      /*Do operation on the data set WORK.&dset.*/
      PROC PRINTTO;
%mend;
%Do_Operation(dataset=First_Dataset);
%Do_Operation(dataset=Second_Dataset);
%Do_Operation(dataset=Third_Dataset);
```

The above code will overwrite the Example_Log.txt external Log file with the Log messages generated by each call of the Macro function %Do_Operation. Therefore, if the external Log file is viewed after all three

Macro calls have been performed, the SAS Log messages for the first two Macro calls will be inaccessible.

However, if the %Do_Operation Macro was modified to include at the end a nested Macro call to a SAS Log parsing Macro %Parse_Log that reported on the content in Example_Log.txt, the Log could be parsed before it was overwritten by the next %Do_Operation Macro. Further, the report generated by such a Macro could associate the extracted Log messages with the value of the %Do_Operation Macro parameter 'dset' if it were passed onto the %Parse_Log Macro call using a Macro parameter for %Parse_Log. Perhaps a third column could be added to the SAS Log report data set shown in Figure 3 that names which 'dset' the reported Log messages were related to.

This method could be thought of as intermediate SAS Log parsing, or parsing of Log information during the execution of a number of processes, rather than at the end of the entire SAS program.

REFERENCES

Stojanovic, Milorad, 2008 "SAS® Log Summarizer – Finding What's Most Important in the SAS® Log"

Proceedings of Southeast SAS User Group Conference 2008. Available at

https://analytics.ncsu.edu/sesug/2008/CC-037.pdf

Hughes, Troy Martin, 2014 "Why Aren't Exception Handling Routines Routine? Toward Reliably Robust Code through Increased Quality Standards in Base SAS"

Proceedings of Midwest SAS User Group Conference 2014. Available at

https://www.mwsug.org/proceedings/2014/BB/MWSUG-2014-BB17.pdf

ACKNOWLEDGMENTS

The author would like to thank NORC at the University of Chicago, Jeff Vose and David Trevarthen for making the development and presentation of this paper at WUSS 2022 possible. Additionally, the author thanks Joe Matise for continued mentoring in SAS and guidance on writing.

CONTACT INFORMATION

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

Drew Metz
NORC at the University of Chicago
55 E Monroe
Chicago, IL 60603
metz-drew@norc.org