Paper SM15

Smarter SAS logging—with a macro AI hasn't mastered (yet)

Jules van der Zalm, OCS Life Sciences, 's-Hertogenbosch, the Netherlands

PREFACE

If you came here for the macro, then scroll all the way down. You can copy and paste it from there and use it as-is. Please keep my and my company's name in there as an acknowledgement.

ABSTRACT

Program logs are vital for debugging code in any language. While SAS® generates solid log files, they're not always great for pinpointing the exact location of problematic code—especially when it's inside a macro.

To facilitate easier code debugging, it's good practice for SAS macros to announce themselves in the log. By printing their name, it becomes much simpler to trace the source of an error message. All you have to do is to scroll upwards in the log and find the latest macro announcement. These announcements often include the macro name as well as parameter names and values.

These announcements are embedded in the program code. Maintaining these announcements is tedious, error-prone and outright distracting. They're usually a series of `%put` statement with the macro name and parameters hardcoded into them. They're often copied from program to program, leading to inconsistencies between what's logged and what the macro is actually doing.

In this paper, I'll present (and share!) a SAS macro that automates this process. Just call it at the top of your macro, with no extra parameters, and your macro will log its own name and parameter values automatically. As icing on the cake, this macro is not (yet) something you can ask AI to create for you!

INTRODUCTION

This is a paper on a very traditional topic in SAS programming in the pharmaceutical and life sciences industry. It's aimed at SAS programmers, novice or expert, that deal with programs of a certain level of complexity that become increasingly difficult to debug. These programs benefit from markers or anchors in the log that help the developer or tester pinpoint the culprit of a bug in case it's a macro or submacro.

Programs can be complex for a number of reasons. They can be single long programs that include a ton of functionality. They can be a combination of one main programming calling a series of macros that provide additional functionality. Or they can be a series of programs, as is often the case in clinical trial analyses where multiple programs are run in series. If programs are made data-driven then the same macro may be looped numerous times, adding to the complexity of debugging.

This paper will talk about a simple way to make the log files of complex (and simple) programs easier to understand. To make reading this paper easier, SAS code is marked with an orange bar in the left margin, while the log is marked with a green bar.

THE TRADITIONAL APPROACH

The traditional approach to announcing a macro in the SAS log is to use `%put` statements to write the macro name and macro parameters with their values to the log. Think along the lines of the following:

```
  /*------------------------------------------------------------------------
    Output macro parameters.
  ------------------------------------------------------------------------*/
  %put ========================================================================;
  %put  Macro readxls;
  %put ------------------------------------------------------------------------;
  %put  The following macro parameters were used:;
  %put - DIR     : &dir.;
  %put - FILE    : &file.;
  %put - SHEET   : &sheet.;
  %put - OUTDS   : &outds.;
  %put ========================================================================;
  %put;
```

For this paper, we will be using an example macro called %readxls. This macro provides a universal way to read various types of Excel files. It's not unthinkable that every company (and perhaps even every program) has some version of such a macro floating around somewhere. In my example this macro takes 4 input parameters:

- **dir**: directory of the Excel file
- **file**: the filename including the extension
- **sheet**: the name of the sheet or tab to read
- **outds**: the name of the output dataset

Including the example code from above, the full macro *and* a call to the macro would look like this.

```
%macro readxls(dir=, file=, sheet=, outds=);

  /*------------------------------------------------------------------------
    Output macro parameters.
  ------------------------------------------------------------------------*/
  %put ========================================================================;
  %put  Macro readxls;
  %put ------------------------------------------------------------------------;
  %put  The following macro parameters were used:;
  %put - DIR     : &dir.;
  %put - FILE    : &file.;
  %put - SHEET   : &sheet.;
  %put - OUTDS   : &outds.;
  %put ========================================================================;
  %put;

  /* Macro magic happens here for the next 100 lines or so... */

%mend readxls;


%readxls(dir  =X:\ONCO\PH-2025-XYZ\Metadata
        ,file =mapping.xlsx
        ,sheet=MappingRules
        ,outds=work.mapping);
```

If this is your SAS code, then your log will look like this:

```
======================================================================
Macro readxls
----------------------------------------------------------------------
The following macro parameters were used:
- DIR     : X:\ONCO\PH-2025-XYZ\Metadata
- FILE    : mapping.xlsx
- SHEET   : MappingRules
- OUTDS   : work.mapping
======================================================================
```

As introduced earlier in this paper, we're calling this the "macro announcement". Clearly, this approach to the macro announcement works. It's simple, it's fairly solid, it's proven. But it's also very prone to mistakes and easily gets out-of-sync with the reality of the macro. It requires updating when the macro changes, and it's prone to the copy-paste-bug that we all get at times.

THE SOLUTION

This step takes you directly to the functionality of the solution. The explanation follows after. My solution to this is a new macro that fully automates announcing the actual macro in the log. The full macro is included at the end of this paper. It revolves around roughly two concepts:

- SAS system macros `%SYSMEXECNAME` and `%SYSMEXECDEPTH`
- SASHELP dataset (or actually, view) VMACRO

Ultimately, the macro `%readxls` will now look like this. Note that the entire block of `%put` statements for the announcement has been replaced by a single macro call to `%logPrintMacroStart` with no parameters.

```
%macro readxls(dir=, file=, sheet=, outds=);

  %logPrintMacroStart();

  /* Macro magic happens here for the next 100 lines or so... */

%mend readxls;
```

The next few paragraphs will go through each component of the solution, which ultimately result in the definition of the macro `%logPrintMacroStart`.

SYSTEM FUNCTION SYSMEXECNAME

`%SYSMEXECNAME` accepts one parameter: a number. When called from a macro, `%SYSMEXECNAME(0)` always returns "`OPEN CODE`". Note: the 0 as the parameter value is important here.

```
%macro readxls;

  %put %sysmexecname(0);

%mend;

%readxls;
```

```
OPEN CODE
```

Now, if you're inside a single macro (as in, the macro is not called from within another macro) and use value 1 for the parameter, it returns the name of the macro:

```
%macro readxls;

  %put %sysmexecname(1);

%mend;

%readxls;
```

```
READXLS
```

Taking this a step further, this is what happens when the macro is called *from within another macro* and value 2 is used for the parameter to `%SYSMEXECNAME`. Note that we're first calling our `%readxls` macro, and `%readxls` uses `%subroutine` as a submacro.

```
%macro subroutine;

  %put Parameter value 2, returns %sysmexecname(2);
  %put Parameter value 1, returns %sysmexecname(1);
  %put Parameter value 0, returns %sysmexecname(0);
  %put This is the subroutine (hardcoded).;

%mend subroutine;

%macro readxls;

  %subroutine;

%mend;

%readxls;
```

```
Parameter value 2, returns SUBROUTINE
Parameter value 1, returns READXLS
Parameter value 0, returns OPEN CODE
This is the subroutine (hardcoded).
```

What this tells us is that the parameter to `%SYSMEXECNAME` tells the SAS macro the "depth" of the macro of which it should return the name, 0 being `OPEN CODE`, any higher value being one level deeper into the macro nesting.

SYSTEM FUNCTION SYSMEXECDEPTH

This logic we can use to make sure that it's always the *actual current macro name* that is printed in the log, if we know how many levels deep we're nested. That is where `%SYSMEXECDEPTH` comes in. This system macro returns the nesting depth of the macro from which it is called; 0 being open code. 1 being within the first macro, 2 is a macro within that macro, et cetera.

```
%macro subroutine;

  %put Value for SYSMEXECDEPTH is %sysmexecdepth;
  %put This is the subroutine (hardcoded).;
  %put Combining the two gives us %sysmexecname (%sysmexecdepth);

%mend subroutine;

%macro readxls;

  %subroutine;

%mend;

%readxls;
```

```
Value for SYSMEXECDEPTH is 2
This is the subroutine (hardcoded).
Combining the two gives us SUBROUTINE
```

These two ingredients combined, so passing `%sysmexecdepth` as a parameter to `%sysmexecname`, gives us the name of the macro *that we're in currently*.

DOING A FULL STACK TRACE

From what we have now, it's a simple step to providing a full stack trace from the deepest macro all the way up to `OPEN CODE`. This helps the reader of the log understand not only which macro we're in currently, but also from which macro it was called and how deeply that was nested into other macros. Rather than *reading* the code below I would recommend to copy and paste it into a SAS session and see it live in action.

```
/* This macro reads a single column. */
%macro readsinglecolumn(file=, tab=, column=);

  %do i = %sysmexecdepth %to 0 %by -1;
    %put %sysfunc(repeat(%str(-), &i)) %sysmexecname(&i);
  %end;

%mend readsinglecolumn;

/* This macro reads a single tab. */
%macro readsingletab(file=, tab=);

  %readsinglecolumn(file=&file., tab=&tab., column=label);

%mend readsingletab;

/* This macro reads a single Excel file. */
%macro readsinglefile(file=);

  %readsingletab(file=&file., tab=Metadata);

%mend readsinglefile;

/* This is the main macro. */
%macro readxls;

  %readsinglefile(file=P:\spreadsheet.xlsx);

%mend;

%readxls;
```

The result is the following list, which perfectly shows the full stack of macros that were called and entered.

```
-----READSINGLECOLUMN
----READSINGLETAB
---READSINGLEFILE
--READXLS
-OPEN CODE
```

SASHELP DATASET VMACRO

So far, we've only been able to print the macro name in the log. A useful achievement, but it essentially serves only a single line in our desired log output: printing the macro name (okay, and a full stack trace if desired). Next, we're going to obtain the macro's parameters and their provided values. This will allow us to print the parameter/value pairs in the log dynamically so we don't have to worry about having the right `%put` statements in the SAS code.

To achieve this, we will be using the dataset (or actually: view) SASHELP.VMACRO. If you look at that dataset at runtime during the deepest macro – in our example, that's `%readsinglecolumn` – it looks like this:

	scope	name	offset	value
1	READSINGLECOLUMN	COLUMN	0	label
2	READSINGLECOLUMN	FILE	0	P:\spreadsheet.xlsx
3	READSINGLECOLUMN	TAB	0	Metadata
4	READSINGLETAB	FILE	0	P:\spreadsheet.xlsx
5	READSINGLETAB	TAB	0	Metadata
6	READSINGLEFILE	FILE	0	P:\spreadsheet.xlsx

Let's break this down:

- The first three records are for the deepest macro, which gets passed parameters `column`, `file`, and `tab`.
- Records 4 and 5 are for `%readsingletab`, our 2nd deepest macro, which gets passed only `file` and `tab`.
- The last record is for `%readsinglefile`, which gets passed only 1 parameter, which is `file`.
- Not shown in this screenshot are tens of rows of automatic variables that are global to your SAS session, which we will not be needing.

We've already learned to extract the name of the macro that we're in, so we have all the ingredients to extract the parameter names *and their values* from the VMACRO dataset. The most obvious and straightforward way would be to do a `PROC SORT` or a `DATA` step, subsetting VMACRO and printing the values. However, that will not work if this or any of the higher-up macros do their macro calls from within a `DATA` step. To circumvent that, we're going to use SAS file I/O functions such as `OPEN`, `FETCH`, `GETVARC`, and `CLOSE`. These work in either situation: in macros called from open code or other macros, and in macros called from within a `DATA` step.

```
/* Get the name of the current macro. */
%let name=%sysmexecname(%sysmexecdepth);

/* Open the VMACRO dataset and fetch the first record. */
%let dsid=%sysfunc(open(sashelp.vmacro));
%let rc=%sysfunc(fetch(&dsid.));

/* Loop through all records in SASHELP.VMACRO and print the parameter
   name and value if it belongs to the calling macro. */
%do %while(&rc=0);
  %let name_ds=%sysfunc(getvarc(&dsid., 1));

  /* If SCOPE matches the macro name then proceed. */
  %if %lowcase(&name_ds.) = %lowcase(&name.) %then %do;

    /* Get the parameter name and its value... */
    %let mvar=%sysfunc(getvarc(&dsid., 2));
    %let mvalue=%sysfunc(getvarc(&dsid., 4));

    /* ... and write them to the log. */
    %put %str(&char2.   ) &mvar. %sysfunc(repeat(%str( ),%eval(20-%length(&mvar.)))) = &mvalue.;
  %end;

  /* Fetch the next record and proceed. */
  %let rc=%sysfunc(fetch(&dsid.));
%end;

/* Close the dataset when done. */
%let qid=%sysfunc(close(&dsid.));
```

This bit of macro code fetches records from VMACRO one-by-one, checks if it relates to the current macro, and if that's the case, it prints the parameter and value pair to the log. This is where things get funny though. If you do this *inside the actual macro* (as the example does) then it will certainly output the macro parameters, but it will also show the macro variables that the code itself created along the way, that we're not interested in (in the example that would be `&dsid`, `&rc`, `&name_ds`, `&mvalue`, and `&mvar`).

Now the *actual solution* (our `%logPrintMacroStart` macro) has this new bit of macro code *inside a submacro* which then fetches the data from VMACRO for "current macro minus one" – which is the macro *from which* `%logPrintMacroStart` was called. This results in a perfectly clean results with only the actual macro parameters, as long as `%logPrintMacroStart` is the very first thing that gets called in the macro.

These ingredients combined result in a macro that does exactly what we're looking for, and that perfectly keeps the macro announcement up to date if the macro names or the parameters change.

DOING THIS WITH ARTIFICIAL INTELLIGENCE

The pace at which programming with AI gets better is staggering. A year ago you could barely get ChatGPT to provide a working program, while today AI will write moderately advanced code *and* the accompanying documentation. There's a fairly large list of programming languages that are well-supported by many LLMs, but SAS is not on that list. You can use LLMs to generate SAS code just fine, but things will go south fast when complexity increases or when the solutions become more arcane.

ATTEMPTS USING WEB-BASED GPTS (CHATGPT, GEMINI)

At the time of writing the abstract for this paper (April 2025), I've used ChatGPT (OpenAI), Gemini (Google), and Claude (Anthropic) to try and reproduce the macro that I already had. Neither of these were able to provide code that would work in any way. They all dream up SAS macro variables and functions that do not exist in reality, and have no working alternative – so just debugging the code and correcting the syntax would not be enough.

The prompt I have used it this:

> *"You're a SAS programmer. Your expertise is SAS Base and macro programming. You're a technical developer with experience in the pharmaceutical industry. You're assigned the task to develop a suite of SAS macros to read, process, and report on data. One requirement is excellent understandability of the SAS log file. Every line in the log should be traceable to the macro that wrote it. Especially if there are warnings or errors it's important that the source macro can easily be identified. To achieve this, each macro announces its start and end in the log. Write a SAS macro with purely this purpose: to report information on the macro that calls it. The information should include at least the name of the calling macro, its parameters and their values, and the macro or macros from which it is called all the way up to the root of the program."*

ATTEMPTS USING CODING ASSISTANTS (ROO CODE WITH CLAUDE)

Coding assistants generally do this job a lot better than LLMs do through their chat interface. Coding assistants such as Roo Code can be configured to work with a multitude of models including those that are optimised for programming, whereas "chats" are more generalistic. Attempts to produce this macro with Roo Code on Claude Sonnet 3.7 produced much better results. The *structure* of the macro that it produces is very close to the structure of the final product of this paper, but this solution too dreams up SAS syntax and makes silly mistakes, such as using `%SYSMEXECNAME` as a macro variable, so referring to it as `&SYSMEXECNAME`. Here, too, fixing these errors still does not result in a working macro.

ARE WE SAFE?

I've personally been deeply impressed by the ability of coding agents to produce working code. In my case that has always been in Python. For these models to work well they need to ingest absolute vast amounts of sample codes, which happen to exist in the public domain for Python and many other languages. We happen to use a programming language (SAS) that is highly proprietary in an industry that is still withholding a lot of their intellectual property; SAS is not popular among those who like to share their code. There's not much sample code available for LLMs to train on. Certainly not as much as there is Python or R code.

Are we safe? That depends on the definition. To those that hope for job security because they happen to use a programming language that AI can't do well: I don't think we're safe. These models will improve, as will their ability to review and improve on their own work. Furthermore the increasing use of R and Python puts us at a disadvantage because AI can do that very well.

I personally like to think that we're safe because we should *embrace* the use of AI. What's better than having a peer programmer that you can provide assignments in little bits and pieces, and that comes back with working code? It's like having an infinite amount of Lego blocks and you can focus fully on putting them together, rather than endlessly trying to find the right piece of Lego.

CONCLUSION

The solution proposed in this paper provides a much simpler and easier-to-use way to have your SAS log state the name of the macro, and its parameters. At the time of writing this paper, AI is not quite there yet with its SAS programming skills and is not able to produce a working solution like this. The SAS macro included at the end of this paper is a fully working solution that anyone is free to use and modify as-is. Please keep attributions to the developer and his company in there.

CONTACT INFORMATION

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

Author Name:	Jules van der Zalm
Company:	OCS Life Sciences (a brand of OCS Consulting)
Address:	Ruwekampweg 2-g, 5222 AT, ‘s-Hertogenbosch, the Netherlands
Work Phone:	+31 73 523 6000
Email:	jules.vanderzalm@ocs-consulting.com
Website:	https://www.ocs-lifesciences.com

APPENDIX 1 – THE FULL MACRO

```
/*************************************************************************
  Copyright OCS Life Sciences
*************************************************************************
  Program     : logPrintMacroStart.sas
  Author      : Jules van der Zalm
  Location    : [macro location here]
  Date (ISO)  : 2025-10-29
  Description : Output the macro name, the macro parameters and the values
                the macro parameters resolved into. This essentially eliminates
                the need to have each macro print its own information frame in
                the log.
  Remarks     : -
*************************************************************************
  Parameters  : None
  Sample call :

    %macro report(inds=, var=, outds=);

     %logPrintMacroStart;

     data &outds;
       set &inds;
       keep &var.;
      run;

    %mend report;

    report(inds=sashelp.class, var=name, outds=work.class);

*************************************************************************
  Modification history:
  Date (ISO)   User    Description
  2025-10-29   Jules   Final version for PHUSE
*************************************************************************/

%macro logPrintMacroStart();

  %local name width char1 char2 trace i dsid rc name_ds mvar mvalue qid total_time;

  /* Find the name of the macro that called logPrintMacroStart
     and store it in &NAME. for later use. */
  %let name=%sysmexecname(%sysmexecdepth-1);

  /* This is the configuration of the frame that is printed in
     the log. The frame looks like this:

        #============================================================
        #  This is the start of macro DO_SOMETHING.
        #============================================================
        #  Stack trace:
        #    -DO_SOMETHING
        #     -FINE_MORE_DATA
        #      -MAIN_MACRO
        #       -OPEN CODE
        #============================================================
        #  Parameters used:
        #
        #    PARAM1                 = A TEXT
        #============================================================

     The setting of WIDTH determines the width of the frame.
     The setting of CHAR1 determines the character of the horizontal lines.
     The setting of CHAR2 determines the character of the left vertical line.
  */
  %let width=60;
```

```
  %let char1=%str(=);
  %let char2=%str(#);

  /* Write to the log "This is the start of macro [macro name]"*/
  %put %str(&char2.)%sysfunc(repeat(&char1., &width.));
  %put %str(&char2.  This is the start of macro %upcase(&name.).);

  /* Write the stack trace, which shows from which macro the macro
     was called (and from which macro _that_ was called, et cetera). */
  %put %str(&char2.)%sysfunc(repeat(&char1., &width.));
  %put %str(&char2.  )Stack trace:;
  /* Loop through all calling macros until OPEN CODE is reached. */
  %do i =0 %to %eval(%sysmexecdepth-1);
    %let trace=%sysmexecname(&i);
    %put %str(&char2.  )%sysfunc(repeat(%str( ), %eval(&i.)))-%upcase(&trace.);
  %end;

  /* Write the parameters and values of the calling macro. This uses
     SASHELP.VMACRO. This step does not use a DATA step so that this
     macro can also be used in macros that are used inside DATA steps.
     This step uses system functions to approach and read the VMACRO
     dataset and find the records that contain the parameters and
     values that belong to the calling macro. */
  %put %str(&char2.)%sysfunc(repeat(&char1., &width.));
  %put %str(&char2.  Parameters used:);
  %put %str(&char2. );

  %let dsid=%sysfunc(open(sashelp.vmacro));
  %let rc=%sysfunc(fetch(&dsid.));

  /* Loop through all records in SASHELP.VMACRO and print the parameter
     name and value if it belongs to the calling macro. */
  %do %while(&rc=0);
    %let name_ds=%sysfunc(getvarc(&dsid., 1));
    %if %lowcase(&name_ds.) = %lowcase(&name.) %then %do;
      %let mvar=%sysfunc(getvarc(&dsid., 2));
      %let mvalue=%sysfunc(getvarc(&dsid., 4));
      %put %str(&char2.   ) &mvar. %sysfunc(repeat(%str( ),%eval(20-%length(&mvar.)))) = &mvalue.;
    %end;
    %let rc=%sysfunc(fetch(&dsid.));
  %end;

  %let qid=%sysfunc(close(&dsid.));

  %put %str(&char2.)%sysfunc(repeat(&char1., &width.));

%mend logPrintMacroStart;
```