Paper CT10

Want to be SASsy?

Parthasarathi Kongara, Merck Specialities Pvt. Ltd., Hyderabad, India

ABSTRACT

Anchor: Want to be SASsy? You're in the right place! Our tricksters are ready to shine. Let's invite our chief guests to the floor!

Trickster 1: I can give you basic visualization capabilities and can plant a tree for you as well. Plus, I can make feedback a breeze. Guess? – PROC SQL

Trickster 2: If you're tired of hacking away through the code jungle, I'm your lifesaver! I'm the automatic code writer here to zap away those tedious programming chores and make your life a whole lot easier!

Trickster 3: Alright, listen up! I've got one last mission—spotting vulnerabilities and enhancing error handling. Get ready to supercharge your code to tackle anything life throws your way. Let's boost productivity and keep everything rock solid!

Anchor: We've introduced few tricksters, but many more are coming to engage you, along with some limitations they'll discuss!

INTRODUCTION

The Statistical Analysis System (SAS) is widely recognized as a robust platform for data management, statistical analysis, and reporting in regulated and non-regulated environments alike. While most practitioners are familiar with its core data step and procedure workflows, a number of powerful but lesser-known features remain underutilized in day-to-day programming. These features can significantly improve developer productivity, simplify debugging, and enhance the transparency and robustness of analytical pipelines.

This paper highlights several such capabilities, with a particular focus on the SQL procedure and selected system options. First, we demonstrate how PROC SQL can be leveraged beyond traditional querying to support tasks such as generating simple bar graphs and using macro variables as lightweight diagnostics for debugging and monitoring intermediate results. We then explore diagnostic options within PROC SQL, such as _METHOD and _TREE, that expose the underlying query execution strategy and join resolution, enabling programmers to understand and optimize query performance with far greater precision.

In addition, we discuss a range of SAS system options—INVALIDDATA, DKRICOND, DKROCOND, and DATASTMTCHK—that can be used to tighten error handling, enforce stricter data quality checks, and detect potential coding issues at an early stage in development. Finally, we show how keyboard macros in Base SAS can be recorded and reused to automate repetitive development tasks, streamlining interactive programming. By bringing these features together, the paper aims to showcase practical techniques that help SAS users write more reliable, transparent, and efficient code with tools that are already available in the base environment but are often overlooked.

POWER OF PROC SQL

Did you know that you can instantly create graphs with PROC SQL? Below PROC SQL step uses a simple text-based technique to visualize frequencies directly in the output window. The GROUP BY sex clause aggregates the rows in sashelp.class by gender, and COUNT(*) computes the number of observations for each group. The REPEAT('*', count(*)-1) expression then creates a horizontal "bar" made of asterisks whose length is proportional to the group count, effectively turning the query result into a rudimentary bar chart. By ordering the result with ORDER BY Count DESC, the category with the highest frequency appears first, making it easy to compare the number of males versus females without needing any dedicated graphical procedures.

```
proc sql;
 select sex, '|', count(*) as Count,
 repeat('*',count(*)-1) as Bar
 from sashelp.class
 group by sex
```

```
 order by Count desc;
quit;
```

Sex	Count	Bar
M	10	**********
F	9	*********

Here is another example of a similar code that shows a simple bar graph.

```
proc sql;
 select age, '|', count(*) as Count,
 repeat('*',count(*)-1) as Bar
 from sashelp.class
 group by age
 order by Count desc;
quit;
```

Age	Count	Bar
12	5	*****
15	4	****
14	4	****
13	3	***
11	2	**
16	1	*

OPTIONS AND MACRO VARIABLES OF PROC SQL

This example illustrates how PROC SQL can be turned into a powerful diagnostic tool rather than just a query engine. By specifying the _METHOD and _TREE options on the PROC SQL statement, SAS prints detailed information to the log about how it plans and executes the SQL step. _METHOD shows the sequence of internal methods and access strategies used (for example, which indexes or join techniques are chosen), while _TREE displays the logical query tree, revealing how SAS parses and optimizes the statement.

In this case, the SQL step performs an UPDATE on the table chk, incrementing the age column by 1 for the observation where name = "Alfred". Although the data manipulation itself is simple, the additional diagnostics provided by _METHOD and _TREE, together with OPTIONS MSGLEVEL=I; (which allows more informative notes and messages in the log), help you understand exactly how SAS is processing the statement. This is especially useful for debugging, performance tuning, and explaining SAS's behavior when working with more complex queries or larger datasets.

```
OPTIONS MSGLEVEL=I;
PROC SQL _METHOD _TREE;
   /* Update age based on average score */
   UPDATE chk AS c
   SET age = age + 1
   WHERE name = "Alfred";
QUIT;
```

```
NOTE: SQL execution methods chosen are:

      sqxupd
          sqxfil
              sqxsrc( WORK.CHK(alias = C) )

Tree as planned.
                                  /-SYM-R-(#RID_PTR:1 flag=00000001)
                        /-OBJ----|
                       |          \-SYM-U-(age:3 flag=00000031)
             /-FIL----|
            |          |                    /-SYM-R-(#RID_PTR:1 flag=00040001)
            |          |          /-OBJ----|
            |          |         |          \-SYM-V-(c.Age:3 flag=00040001)
            |          |--SRC----|
            |          |         |--TABL[WORK].chk opt=''
            |          |         |          /-NAME--(Name:1)
            |          |          \-CEQ----|
            |          |                    \-LITC('Alfred')
            |          |--empty-
            |          |--empty-
            |          |--empty-
            |          |--empty-
            |          |                    /-SYM-U-(age:3 flag=00000031)
            |          |          /-ASGN---|
            |          |         |          |          /-SYM-V-(c.Age:3)
            |          |         |          \-AADD---|
            |          |         |                    \-LITN(1)
            |           \-OBJE---|
 --SUPD---|
```

Below are a few codes that are represented by _METHOD.

Codes	Description
sqxcrta	Create table as Select
sqxslct	Select
sqxjsl	Step loop join (Cartesian)
sqxjm	Merge join
sqxjndx	Index join
sqxjhsh	Hash join
sqxsort	Sort
sqxsrc	Source rows from table
sqxfil	Filter rows
sqxsumg	Summary stats with GROUP BY
sqxsumn	Summary stats with no GROUP BY

In addition to the above options, there are a few automatic macro variables that are useful in debugging a code. These automatic macro variables provide a compact but powerful way to debug, and monitor PROC SQL execution. SQLRC returns a status code for each SQL statement, allowing you to programmatically distinguish between

successful runs (0), warnings (4), errors (8), and more severe conditions such as internal, user, or system errors (codes 12, 16, 24, and 28). SQLOOPS reports how many times the inner loop of PROC SQL executed, which is particularly useful for understanding the complexity and efficiency of joins or nested queries. When working with SQL Pass-Through, SQLXRC and SQLXMSG expose the database-specific return code and its descriptive message, making it easier to diagnose problems originating in the external DBMS rather than in SAS itself. Together, these macro variables enable more informative logging, conditional error handling, and automated checks that greatly simplify debugging in complex SAS/SQL programs.

ERROR HANDLING

Robust error handling is an essential yet often overlooked part of SAS programming, and SAS provides several mechanisms to help us move beyond simply reacting to log messages. By combining diagnostic tools, automatic macro variables, and system options, we can detect issues earlier and control how the system responds to them. In the example above, a deliberate error is created by referencing a non-existent data set in a DATA _NULL_ step, and then the macro variables &SYSERR and &SYSERRORTEXT are checked immediately afterward. Instead of letting the error scroll by unnoticed in the log, the code tests whether &SYSERR is non-zero and, if so, prints a clear, customized message using PUT in the log. This pattern illustrates how we can capture runtime failures programmatically and turn them into explicit, meaningful feedback, making our programs more transparent, reliable, and easier to debug.

```
data _null_;
   /* Create an error by referencing a non-existent dataset */
   set non_existent_ds;
   newvar = height + 10;
run;
/* Check for errors */
%if &syserr ne 0 %then %do;
   %put ERROR: An error occurred during the data step. &syserrortext;
%end;
```

While macro variables such as &SYSERR and &SYSERRORTEXT allow us to detect and react to errors programmatically, there are situations where we may want SAS to halt immediately when a serious problem occurs. The ERRORABEND system option supports this stricter style of error handling by terminating the SAS session as soon as an error is encountered. In the example below, attempting to read from a non-existent data set within the DATA step triggers an error, and ERRORABEND ensures that processing stops right at that point instead of continuing with subsequent steps. This approach is especially useful in production or regulated environments, where silently skipping over failures is more dangerous than stopping the job and demanding intervention.

```
options errorabend; /* Set the system option */
 data test;
 /* Create an error by referencing a non-existent dataset */
    set non_existent_ds;
    newvar = height + 10;
 run; /* This will stop execution due to the error */
```

These two examples illustrate complementary approaches to error handling in SAS: defensive checks at the macro level and observation-level handling in the data step. In Code 1, the %EXPLORE macro validates its inputs up front by ensuring that both INPUTDATA= and VAR= are provided; if either is missing, it writes a clear error message to the log and uses %ABORT CANCEL to stop execution, preventing the macro from running with incomplete or incorrect parameters. In Code 2, error handling is embedded directly in the data step through the automatic _ERROR_ variable: when an invalid operation (such as division by zero) occurs, _ERROR_ is set, and the code immediately logs a descriptive message with the offending observation number and name, resets _ERROR_ to allow processing to continue, and replaces the problematic result with a missing value. Together, these techniques help catch issues early, provide targeted diagnostics, and control whether processing should halt or continue gracefully in the presence of errors.

Code 1:

```
%macro explore(inputdata=, var=);
    %if %length(&inputdata) = 0 %then %do;
      %put ERROR: INPUTDATA= must be specified;
%abort cancel;
    %end;
    %if %length(&var) = 0 %then %do;
```

Code 2:

```
data test;
  set sashelp.class;

  /* Intentionally create an error */
  result = height / 0;  /* Division by zero */
```

```
      %put ERROR: VAR= must be specified; %abort
cancel;
    %end;
    /* Proceed with the rest of the macro */
  %mend;
```

```
  /* Check for error and take action */
  if _error_ then do;
    put "ERROR detected for observation " _n_ ": "
name=;
    _error_ = 0;  /* Reset error flag to continue
processing */
    result = .;   /* Set to missing value */
  end;
run;
```

INVALIDDATA

These examples show how the INVALIDDATA system option lets you control how SAS responds when it encounters bad numeric values during input. By setting OPTIONS INVALIDDATA=A;, any value that cannot be read as a valid number (such as ABC in the age field for Jane) is automatically converted to the specified code, here A, making invalid entries easy to flag and track in subsequent analyses. This is particularly useful when you want to distinguish between genuinely missing values and values that were present but invalid. In contrast, resetting INVALIDDATA=.; returns to the default behavior, where invalid numeric data are quietly converted to standard missing, blending them with other missing values. Together, these settings give you fine-grained control over how data quality issues are represented and handled in your datasets.

```
/* Example with INVALIDDATA option */
options invaliddata=A;  /* Assign 999 to invalid numeric data */
data test;
   input name $ age;
   datalines;
John 25
Jane ABC  /* Invalid numeric - will be set to A */
Bob 30
;
run;
/* Alternative */
options invaliddata=.;  /* Default - assign missing value */
```

DKR-X-COND

These options demonstrate how the DKRICOND option can tighten error handling around variable lists. With OPTIONS DKRICOND=ERROR;, SAS will generate an error if you reference a variable that does not exist in the input data set. In the example, the DATA NEW_TEST; SET TEST(DROP=WEIGHT); step fails because WEIGHT is not a variable in TEST, and DKRICOND=ERROR prevents SAS from silently ignoring the mistake. This behavior helps catch typos and incorrect assumptions about the structure of a data set early, rather than allowing flawed code to run unnoticed.

```
/* Example with DKRICOND option */
options dkricond=error;  /* Generate error for non-existent variables */
data test;
   input name $ age height;
   datalines;
John 25 70
Jane 30 65
;
run;
data new_test;
   set test(drop=weight);  /* 'weight' doesn't exist in test dataset */
run;
/* This will generate an ERROR because 'weight' variable doesn't exist */
```

While DKRICOND works on DROP, KEEP and REPLACE on the input dataset a similar option “DKRICOND” works on DROP, KEEP and REPLACE when these are mentioned on the output dataset. Values for the option are error, warn and nowarn.

DATASTMTCHK

This example shows how DATASTMTCHK can help catch coding mistakes in DATA step statements. With OPTIONS DATASTMTCHK=ALLKEYWORDS;, SAS checks whether certain words used in DATA statements might be reserved keywords (for example, expecting DATA, SET, MERGE, INFILE, etc.) and flags suspicious usage. In the code above, DATA species INFILE 'bugspecies.dat'; might trigger a diagnostic if SAS interprets INFILE in an unexpected way, prompting you to separate the DATA and INFILE statements correctly. This added scrutiny makes it easier to spot subtle syntax errors or misused keywords before they lead to confusing runtime behavior.

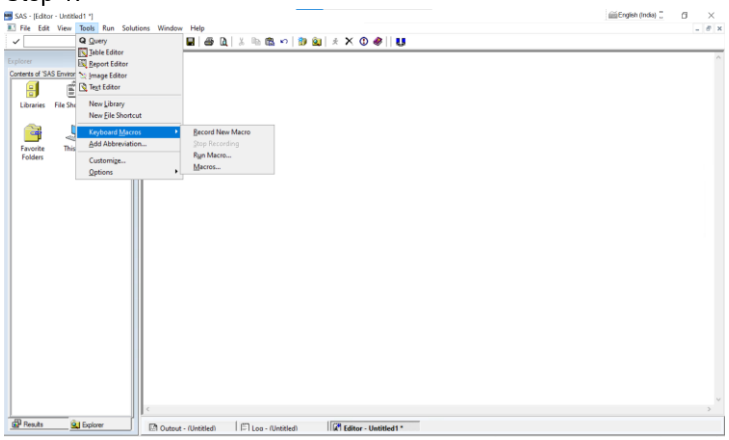
```
OPTIONS DATASTMTCHK=ALLKEYWORDS;
DATA species
INFILE 'bugspecies.dat';
INPUT Order $ 1-15 InNorthAmerica OutsideNorthAmerica;
RUN;
```

CODING MADE EASY

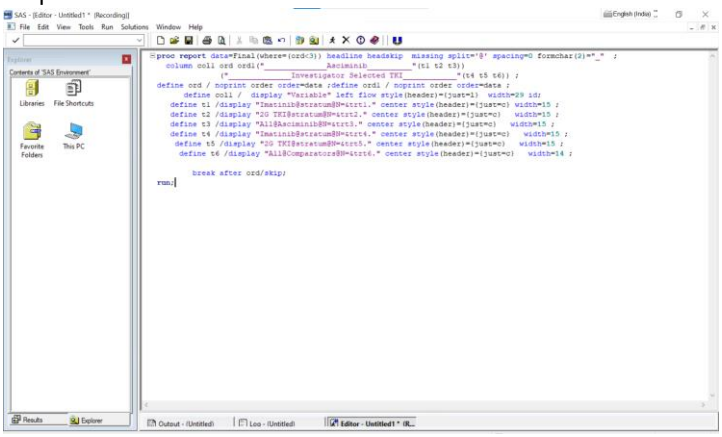
Keyboard macros are a simple but powerful way to automate repetitive actions by recording a sequence of keystrokes and replaying them with a single key or key combination. Instead of manually retyping the same commands or edits, you record the sequence once, assign it to a key, and then invoke it whenever needed. This can dramatically speed up routine tasks such as formatting code, inserting standard templates, or running common commands. Many development environments and editors support both temporary (one-off) and persistent keyboard macros, allowing users to fine-tune their workflows and reduce the risk of manual errors. By leveraging keyboard macros, programmers can focus more on problem-solving and less on mechanical, repetitive typing. Below are the steps in base SAS to record and execute keyboard macros.

Tools>Keyboard Macros>Record New Macro>Type the code>Stop Recording>Name the Macro>Assign Key combo

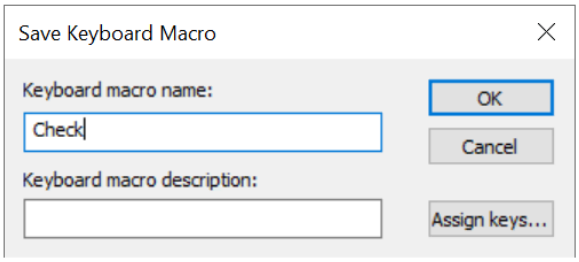
Step 1:



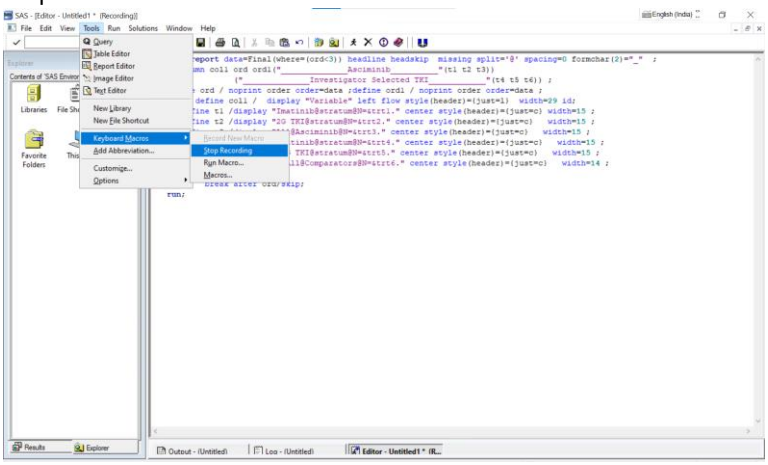
Step 2:



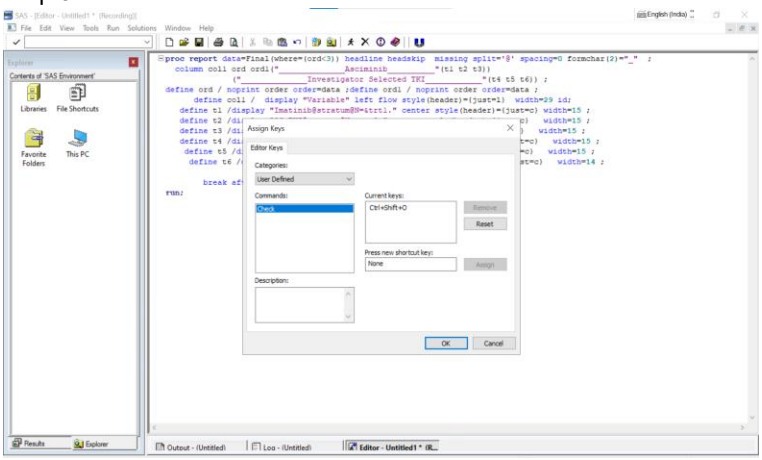
Step 3:



Step 4:



Step 5:



CONCLUSION

Although SAS is often associated with traditional DATA steps and standard procedures, exploring its lesser-known features reveals a rich toolbox for writing clearer, safer, and more efficient programs. Simple techniques—such as using PROC SQL to generate text-based bar charts or leveraging SQL diagnostic options like _METHOD and _TREE—can make our code both more insightful and more transparent. Systematic error handling through automatic macro variables (SQLRC, SQLOOPS, SYSERR, SYSERRORTEXT), system options (ERRORABEND, INVALIDDATA, DKRICOND, DATASTMTCHK), and structured patterns in both macros and DATA steps allows us to detect problems early, decide when to fail fast, and when to recover gracefully. Finally, stepping beyond pure syntax into workflow improvements, tools like keyboard macros help automate repetitive actions and reduce friction in day-to-day development. Taken together, these capabilities demonstrate that much of SAS's real power lies not only in its analytical procedures, but in the subtle options and techniques that, once understood, can transform everyday programming practice.

REFERENCES

https://www.lexjansen.com/wuss/2016/168_Final_Paper_PDF.pdf
https://www.sfu.ca/sasdoc/sashtml/lrcon/z0993436.htm
https://support.sas.com/resources/papers/proceedings/proceedings/sugi30/101-30.pdf
https://communities.sas.com/t5/SAS-Programming/Error-Handling-in-a-construct-like-try-catch/td-p/539151

ACKNOWLEDGMENTS

I would like to express my sincere gratitude to Anik Chatterjee for his invaluable assistance with a few topics in this paper. His insights and support were instrumental in enhancing the quality of my work.

CONTACT INFORMATION

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

Author Name: Parthasarathi Kongara
Company: Merck Specialities Pvt. Ltd.
Email: parthasarathi.kongara@merckgroup.com