PharmaSUG 2025 - Paper AP-033

Four Roads Lead to Outputting Datasets Dynamically (ODDy) & Beyond
A Case Study for SAS® Data-driven Programming

Jason Su, Daiichi Sankyo, Inc.

ABSTRACT

Outputting Datasets Dynamically (ODDy) is a common programming task, which is to partition an input table based on its contents on-the-fly. Although WHERE statement can be used to create subset datasets individually, the variable values to be used in the statement must be known beforehand, and the method is not quite efficient itself. The most efficient way might be to create all with just one-passing of the source or master one with OUTPUT statements. Then the question is: How to automate the whole process and create these OUTPUT statements dynamically? Here three popular methods are provided: MACRO, FILENAME+%INCLUDE, and CALL EXECUTE routine. Additionally, the paper gives two variants of the 1st solution, namely a macro program and a macro function. Finally, besides these generalized solutions, there is a special solution especially for ODDy, which is to output with HASH object. More importantly, from the perspective of data-driven programming, the paper summarizes the advantages/disadvantages and gives short insight on the circumstance niches for of each technique. In the end, the readers will be more familiar with these powerful solutions for data-driven tasks and their primary differences, and in the future when facing other data-driven programming tasks, can immediately pick the best from the four techniques based on specific circumstances.

INTRODUCTION

Dynamic SAS® Programming is a data-agnostic technique to enable your SAS program to automatically generate SAS code based on the data in a control file, so it is also called data-driven programming. It is well known that macros are especially good at this. However, when you are reading your fellow programmers' work, you might discover that other techniques are also employed, such as CALL EXECUTE routine and FILENAME with %INCLUDE. You may be wondering what they are and why they are used there, instead of the powerful macro method. Due to the space limitation and the availability of many excellent papers already on the what-they-are topic [Carpenter 2004], this paper will primarily focus on the difference between these techniques, and stress on the situations where one might be preferable to others.

The best way to learn is to do it through examples and here the issue of Outputting Datasets Dynamically (ODDy) is as an example, which is a common programming task. For starters, we have a simplified dataset with two variables DSET and IDLIST: The former is the name of sub-datasets; The later contains the ids and is the primary data for processing. The dataset is created with a DATA step as follows,

```
data master;
    length dset idlist $20;

    dset='dataset2'; idlist='444';       output;
    dset='dataset3'; idlist='4442,5552'; output;
run;
```

The job has two-fold purpose as follows,

1. breaking up these IDLIST strings based on comma and saving the normalized form into a new variable ID, as being a side task for demonstration.
2. based on variable DSET values, splitting up the source or master dataset efficiently and dynamically into sub-datasets, i.e. without manually inputting these static values in the starting program. The big benefit is that without modification, the same program can still work even though other variants of the dataset master might contain other values such as dataset4, misc100, etc.

We will focus on the second one from now on. An easy solution to it for many is to use the WHERE statement in a loop, with each iteration changing the conditions in the WHERE statement such as

```
dset='dataset2'
```

However, to maximumly reduce the computer IO cost (input/output), we don't prefer to go down that road at all. Rather we want to load the whole dataset only once and all the sub-datasets are created, even though we don't know what the specific values are in DSET variable. This may be the light-speed cap, i.e. as efficient as human beings can do. Is that possible?

The answer is YES, and the OUTPUT statements are definitely the tool for the job. However, after you try a few times, you might realize the sub-dataset names (i.e. these variable DSET values in the example) must be known beforehand, and they must be spelled out in the OUTPUT statement in the compile time. That is to say, this statement will not create these sub-datasets as desired

```
   output dset;
```

Here, our dynamic programming comes into the picture: A series OUTPUT statements combined with the simple IF/THEN logic must be produced first, and then with the rest of the DATA step components, all the sub-datasets will be created with one-passing of the data. That is to say, the goal is to primarily produce something like

```
if        dset='dataset2' then output dataset2;
  else if dset='dataset3' then output dataset3;
```

We now are ready to talk about these three roads and their solutions for our issue at hand first. In the whole, we desire these codes to be created as follows,

```
data dataset2 dataset3;
    length id $10;

    set master;
    do i=1 to countw(idlist);
        id=scan(idlist,i);
        if dset="dataset2" then output dataset2 ;
           else if dset="dataset3" then output dataset3 ;
    end;
    keep id;
run;
```

ROAD #1. THE MACRO

The idea has two steps:

1. First save the data values from the control file into macro variables,
2. And then in macros, integrate them with the SAS statements to run.

Often this may be the solution for most people when talk about dynamic SAS programming. And as we dive further in this area, this is rightly so. Carpenter (2004) in his classic book on MACRO coincidently provides a solution with macro arrays or a series of macro variables with a shared root and a serial number in the macro name. As also suggested by the author, a macro list can also be used where all the values are stashed into a single macro variable. For a change, here the macro list is used for our implementation.

First of all, for ODDy, since these sub-dataset names must appear in the DATA statement we can collect the values into this macro variable LIST like this.

```
proc sql noprint;
    select distinct dset into : list separated by ‘ ‘
    from master;
quit;
```

Then, we can integrate them into this macro as follows,

```
%macro spliting(list=);
    data &list.;
        length id $10;

        set master;

        do i=1 to countw(idlist); ❶
            id=scan(idlist,i);

            %do i=1 %to %sysfunc(countw(&list.)); ❷
                %let ds=%scan(&list.,&i.);
                %if &i.>1 %then else; ❸

                if dset="&ds." then output &ds.;❹
            %end;
        end;

        keep id;
    run;
%mend spliting;

%spliting(list=&list.);
```

❶ Splitting the IDLIST values with COUNTW & SCAN functions and default delimiters.
❷ Likewise splitting the values in &LIST with COUNTW & SCAN functions and default delimiters.
❸ The keyword "else" is added after the first OUTPUT statement.
❹ This is the statement(s) we are looking for. The sub-dataset names can be easily modified, especially when the DSET values are not proper for dataset names directly, such as '1', '4 subdata', etc.

Regarding the performance difference between macro arrays and macro lists there might be nothing detectable. Clearly, there is a maximum length limitation in a macro variable. So, if you don't desire so many macro variables in the system and the total character is within the 65,534 character-limitation, the macro list is your choice as above.

Please note that the sub-dataset names can be

Alternatively, a customary macro function can also be used here, which is a special macro structure where macro statements are exclusively employed, and some values need to be returned at the end of function execution. For ODDy here, we desire these values in form of the OUTPUT statements to be returned when the macro function is called in the DATA step. The macro function with one argument would be as follows,

```
%macro getting_output_stmnt(control_ds);
    %if ^%sysfunc(exist(&control_ds.)) %then %return;

    %let dsid=%sysfunc(open(&control_ds.));
    %let i=0;

    %do %while(%sysfunc(fetch(&dsid))=0);
        %let i=%eval(&i.+1);
        %let ds=%sysfunc(getvarc(&dsid,%sysfunc(varnum(&dsid.,dset)))); ❶
        %if &i.>1 %then else;
        if dset="&ds." then output &ds.; ❷
     %end;
     %let rc=%sysfunc(close(&dsid.));
%mend;

data &list.;
    length id $10;

    set master;

    do i=1 to countw(idlist);
        id=scan(idlist,i);
        %getting_output_stmnt(work.master)  ❸
    end;

    keep id;
run;
```

❶ Directly fetch DSET values from each observation of the dataset master.
❷ Writing out the OUTPUT statements like before. The string is the output of the function.
❸ Calling the macro function and populating the statements. Note that semicolon is not needed here.

It goes without saying that the macro function %getting_output_stmnt itself would get shorter and run slightly faster if the macro variable &LIST were directly employed, instead of the SAS I/O functions, but here I want to provide some additional routes in case that it is really needed. Additionally, advanced programmer might already have detected that the direct fetch method has another minor issue: Any duplicate DSET values such as 'dataset2' in the control file *master* would render the result as

```
if              dset='dataset2' then output dataset2;
  else if       dset='dataset2' then output dataset2;
```

Although they can run without any syntax issues, the created statements definitely are a frown-upon practice for many, let alone the inferior performance if many sub-datasets to be created.

The macro route seems straightforward to deal with the ODDy issue; However, in other data-driven programming situations there might be many data variables from the control file to be incorporated into the final data statement, these macro variables may get mingled and messed up, making the method unwieldy. In that cases, other routes might become advantageous.

ROAD #2. THE CALL EXECUTE ROUTINE

CALL EXECUTE routine is a routine in DATA step and the idea is to directly output and execute SAS codes based on the data from the control file with powerful DATA step functionalities. Since it has only one argument, which could be any character-producing elements including quoted strings, a character expression/function, or a character variable; The last two are the data entry spots for data-driven programming. These SAS codes will be put into the input stack right after the ongoing DATA step. However, if there is some macro component such as statement/invocation in the SAS codes, the current DATA step is temporarily suspended and the macro facility is invoked for dealing the macro component. Therefore the timing issue should be paid much attention wherever there are macro involved in the routine. Fortunately there are plenty of edifying documentations on the subject already and readers are strongly encouraged to delve in.

For our ODDy, it could be solved with two Do-loop-of-Whitlock (DOW) structures since data are needed at two spots as follows,

```
data _null_;
   length dslist $200;

    **CREATING THE DS LIST FOR DATA STATEMENT INSIDE OF DOW;  ❶
    do until (eof);
        set master (keep=dset) end=eof;
```

```
            dslist=catx(‘ ’,dslist,dset);
        end;

        call execute(catx(‘ ’,'data’, dslist,’;’)); ❷
        call execute('    length id $10;');

        call execute('    set master;');

        call execute('    do i=1 to countw(idlist);');
        call execute('          id=scan(idlist,i);');

        **CREATING THE OUTPUT STATEMENTS INSIDE OF DOW; ❸
        do until (eof2);
            set master (keep=dset) end=eof2;

            _num + 1;
             call execute(catx(' ',ifc(_num>1,'else',''),cats('if
dset="',dset,'" then output'),dset,';')); ❹
        end;

        call execute('    end;');

        call execute('    keep id;');
        call execute('run;');
    run;
```

❶ The 1st DOW block for collecting DSET values into variable DSLIST.
❷ CATX function is used here for concatenating the ‘data’ keyword and the sub-dataset names.
❸ The 2nd DOW block for composing the OUTPUT statement with the data from master .
❹ Auto retained temporary variable _NUM combined with the IFC function are to incorporate the keyword ‘else’ into the same CALL EXECUTE routine.

The created SAS statements are available in the LOG unless the system option NOSOURCE is designated. Please note that if there exist same DSET values in the observations from the dataset *master* as pointed out during the macro function implementation as above, two potential issues would arise:

1. Any duplicate DSET values can give you the redundant OUTPUT statements and this can be easily eliminated with a temporary dataset instead of the source dataset master, where only unique DSET values should be present
2. These duplicate DSET values will generate the DATA statement from the 1st DOW block for collecting DSET values like this

   ```
   data dataset2 dataset2 dataset3 ;
   ```

   where the dataset value “`dataset2`” are duplicated. This will fail during the resultant DATA-step execution. A temporary dataset has to be used instead, where the DSET values have been unduplicated.

Therefore, for ODDy, the direct use of the original control file master with CALL EXECUTE routine might not be possible and if indiscrete, the technique itself could land you the errors.

Additionally, CALL EXECUTE routine is used in this example exclusively and in rea life it does not need to be. For example, macro variable &LIST can be much handy and the 1st DOW block can be simply replaced with something like here,

```
call execute('data &list;');
```

Also as Hughes points out, CALL EXECUTE with a user-defined macro might be a much better solution to obliviate the readability issue [Hughes 2022].

In all, since this method directly assembles and executes SAS codes with the data in control files, it can deal with dynamic programming quite easily, even if there are many data variables from the control file to be incorporated into the final SAS codes. I personally prefer the CALL EXECUTE routine to the macro method if there are more than 3 data variables to be used from the control files.

Compared with the MACRO method, there are two disadvantages as follows,

1. Coding and debugging may be a real pain sometimes, since in the originating DATA-step it may be very challenging to check on all the quotation marks mingled with multiple character data values from the control files.
2. As pointed by many experts, timing issue may get you trouble if macro variables have to be created first by the new SAS codes and called subsequently in the codes, and their resolution is not dealt properly. There are numerous papers on this topic already and readers need to be clear on the matter.

ROAD #3. FILENAME+%INCLUDE

FILENAME statement is a statement to associate/dissociate a SAS fileref with an external file/output device, depending on the arguments. For data-driven programming, similar to the CALL EXECUTE routine method, the idea is to use DATA step to create our desired SAS codes with the data from control files. But there are two main differences.

1. The FILENAME further needs a tag-along %INCLUDE statement for execution in the instant program, whereas the CALL EXECUTE routine automatically has the resulting codes executed afterwards, as the routine name indicates.
2. PUT statement is used instead of CALL EXECUTE ROUTINE.

The ODDy issue is solved with this road as follows,

```
filename pgm temp; ❶
/* Create the generated program */
data _null_;
    length string $500;

    file pgm; ❷

    put 'data &list;'; ❸
    put '    length id $10;';

    put '    set master;';

    put '    do i=1 to countw(idlist);';
    put '          id=scan(idlist,i);';

    **CREATING OUTPUT STATEMENTS INSIDE OF DOW; ❹
    do until (eof);
        set master (keep=dset) end=eof;
        _num + 1;
        string = catx(' ',ifc(_num>1,'else',''),'if dset="',dset,'" then 
output',dset,';');
        put string;
    end;

    put '    end;';

    put '    keep id;';
    put 'run;';
run;
/* Run it */
%include pgm/source2; ❺
```

❶ FILENAME statements associates the fileref PGM with a device TEMP, which exists only as long as the filename is assigned. For easy debugging in the initial development, however, a physical address might be preferable since all the created statements can be stored in one place for checking before the final execution.
❷ FILE statement directs the text generated by PUT statements to the fileref PGM.
❸ As mentioned earlier, the macro variable &LIST is employed. Alternatively a DOW can be used for stashing DSET values into a DATA step variable. The PUT statements write out the strings to the target through the fileref.
❹ The DOW incorporates the data from the dataset master into the OUTPUT statements likewise.
❺ %INCLUDE statement executes the contents associated with fileref PGM as a whole.

The method has a sandwich-like structure: The middle section is the DATA step, which is preluded with a FILENAME and closed with an %INCLUDE statement. Different with the CALL EXECUTE routine, PUT statements takes no argument and direct employing of SAS DATA-step functions is prohibited, although the PUT statement has numerous options. It might be a disadvantage since usually the data in the control file have to be modified before incorporation. For example, as in the example above, the DSET value must be stripped or trimmed before quotation marks are added, and we use a temporary variable STRING to solve the issue. Personally, the CALL EXECUTE routine may be much neater in this regard because all these can happen inside of the argument.

Even though, the FILENAME+%INCLUDE method has two advantages over the CALL EXECUTE routine method as follows,

1. All the output SAS codes are in one file, where they can be readily tested and debugged. This is more especially important during the early development. In the CALL EXECUTE routine this file does not exist and therefore a standalone modifying and testing the created SAS codes would not be much accessible, although the created SAS codes can be gleaned from the LOG only *after* the successful execution of the instant DATA-step.
2. It does not have the macro-variable-related timing issue since the macro compiler is not involved during the SAS codes creation. Based on the same reason, this road may be computationally faster surmised by some experts, albeit negligible for modern computers.

Similar to the CALL EXECUTE routine method, coding may be a real pain because the data values, quotation marks and the DATA-step elements are mingled altogether. Also this method enjoys the straightforwardness advantage over the MACRO method if there are many data variables from the control file to be incorporated into the final SAS codes.

ROAD #4. HASH, A SPECIAL METHOD

As Dunn (2008) demonstrated, HASH happens to have such a method called OUTPUT, which fits perfectly into our ODDy scenario. The special solution would be as follows,

```
    data _null_;
        length id $10;

        if _n_=1 then ❶
            do;
                dcl hash h();
                h.definekey('id'); ❷
                h.definedone();
            end;

        set master;

        do i=1 to countw(idlist);
            id=scan(idlist,i);
            rc=h.ref(); ❸
        end;

        rc=h.output(dataset:dset); ❹
        rc=h.clear();
    run;
```

❶ Only create the HASH once. The object is cleared but not removed at the end of each iteration.
❷ DEFINEKEY() alone is enough for this job but normally DEFINEDATA() method is needed.
❸ REF() method is working like ADD() without worrying about the key duplication issue.
❹ The variable value of DSET is used as parameter values in OUTPUT() method. Also, as pointed out earlier, the DSET variable can be modified directly in this parameter, especially if the values are not legitimate dataset names themselves.

Since there is no need to prepare or collect DSET values, one passing of the data can totally solve the problem. Although HASH may seem a natural solution for ODDy, there is no appending-like method available yet. Therefore, if there are multiple records sharing the same DSET values, the OUTPUT method would simply overwrite the dataset from the previous observation without a warning, and only the last observations in the source master data are saved in the sub-datasets. In order for us to prevent this from happening, either a cautionary pre-checking step or a DSET-based data pre-aggregation step might be added. Even so, for ODDy issue alone, the HASH solution might still be the winner due to its brevity of the codes and likely performance supremacy.

SUMMARY: THE SOLUTIONS OF DYNAMIC PROGRAMMING

Now we have seen how these techniques solve the ODDy issue, and I summarize their primary differences in Table 1.

	Advantage	**Disadvantage**
MACRO	1) Relatively easy to code and to read. 2) Powerful and versatile.	If there are too many data variables to incorporate, it becomes unwieldy.
CALL EXECUTE	1) Directly composing the argument with SAS functions. 2) Can handle as many data variables as needed. 3) Stand alone. No other statements needed.	1) Hard to code and debug due to mingled texts and quotation etc. 2) Timing issue if macros are involved.
FILENAME+%INCLUDE	1) Can handle as many data variables as needed. 2) No timing issue. 3) Easier debugging since all codes are output to one place.	1) Hard to code due to mingled texts and quotation etc. 2) PUT statement does not have arguments.
HASH	1) Great for outputting datasets dynamically.	1) Might not be for other data-driven programming.

Table 1. Comparison of the Dynamic Programming Techniques

Additionally, the CALL EXECUTE routine and the FILENAME+%INCLUDE method can create other datasets on the side based on the control files in the same DATA step (instead of _NULL_ as in above examples), besides the SAS statements. The routes probably complicate situations a bit but definitely have the capacity the MACRO method is short of.

CONCLUSION

For the grand picture of SAS data-driven program design, Troy Martin Hughes' recent book [Hughes 2022] might have to be consulted. However, here we are mainly focusing on the fundamental techniques and use outputting datasets dynamically (ODDy) as a case study for data-driven programming. We summarize these following factors for selecting the best techniques in each circumstance:

- How many data variables/elements come from the control files.
  The MACRO method is suggested if few data elements are incorporated. Otherwise, other methods seem more appealing.
- Length of the created SAS statements.
  If lots of SAS statement to be created, the MACRO method comes more easily.
- The data status in control files and final statements to create.
  For example, as in the ODDy, if there are observations with duplicate DSET values in master, only the MACRO method can create handily the DATA statements with the original control file, whereas the OUTPUT statements can be created by all these techniques. On the other hand, for a control file with unique DSET values, all three techniques can be used.

Mostly the decision is based on your knowledge and familiarity with each technique.

Also, as pointed out by Lafler 2008 and many others, the tables/views in DICTIONARY/SASHELP are popular data sources for data-driven programming. But regarding the techniques, besides these main methods, there be others such as PROC TRANSPOSE, PROC FORMAT, ODS OUTPUT, etc. which might be quite advantageous in some special situations. For us, analyzing each of the scenarios and quickly finding the best road may be exactly the spot that makes SAS programming enjoyable each day.

REFERENCES

Lafler, Kirk Paul. 2008. "Data-Driven Programming Techniques Using SAS" *Proceedings of the SAS Global Forum 2021 Conference*. Available at https://communities.sas.com/t5/SAS-Global-Forum-Proceedings/Data-Driven-Programming-Techniques-Using-SAS/ta-p/726293

Dunn, Toby. 2008. "Breaking Up Isn't Hard To Do After All" *Proceedings of the SESUG 2008 Conference*. Available at http://analytics.ncsu.edu/sesug/2008/SBC-119.pdf

Carpenter, Art 2004. *Carpenter's Complete Guide to the SAS Macro Language*. 2nd ed. Cary, NC: SAS Publishing.

https://communities.sas.com/t5/SAS-Programming/output-a-dataset-with-a-dynamic-name-rather-than-staic-name/td-p/74603

Hughes, Troy Martin. 2022. *SAS Data-Driven Development: From Abstract Design to Dynamic Functionality*. 2nd ed. Independently published.

ACKNOWLEDGMENTS

I appreciate my management Yongin Kim, Yoko Tanaka, and Ajay Gupta for their valuable comments. This paper would not be possible if in the absence of their tremendous support & encouragement.

CONTACT INFORMATION

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

Jason Su
Daiichi Sankyo Incorporation
919-260-5649
Jason.su@daiichisankyo.com