Suggestions for Organizing SAS® Code and Project Files

Nate Derby

http://www.nderby.org

September 2, 2010

ABSTRACT

For a beginning SAS® user, there are several resources available for learning how to write efficient programs, but few for how to organize code, program files, data sets, and related files. This is very important for knowing where to find everything, avoiding duplicating efforts, and maintaining version controls. This paper presents some ideas for organizing these files, along with some short and useful pieces of code.

Keywords: Macros, code, organize.

INTRODUCTION

There are many resources available for learning how to write efficient code (e.g., Carpenter (1999); Lafler (2000); Winn Jr. (2004); Chidambaram (2005); Langston (2005); Jolley and Stroupe (2007), or many other entries from an internet search of "writing efficient SAS code"). However, with the notable exception of Marje Fecht and co-authors (Fecht and Stewart, 2008; Dosani et al., 2010), there appear to be few available for organizing code or project files. This paper is not a comprehensive guide to this subject – rather, it is a short overview of one set of ideas. There are many ways to be organized, and the best way depends on many factors: Is the work done by one person or within a team? Does the project have specifications? Is the project ongoing, or does it have a specific deadline? The opinions contained in this paper are from the perspective of a single SAS programmer in a fast-paced business setting with many changing aspects, within a small team of managers, analysts and/or non-SAS programmers.

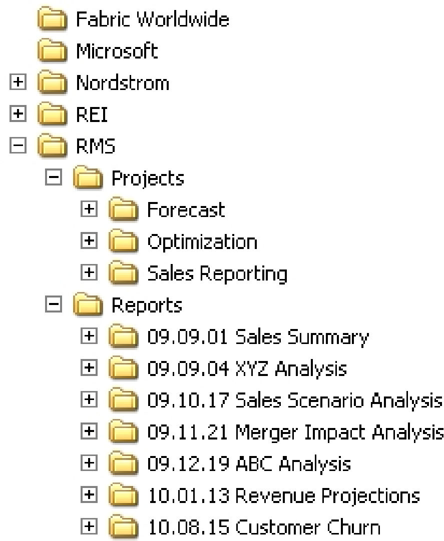
In the business setting described here, many project aspects change constantly – deadlines get moved, one-time reports become projects (or vice-versa), focuses change, and occasionally an obscure report from the distant past suddenly becomes relevant. The ideas presented here are meant to make it easier to deal with these changes. This organizational style has been effective for the author for over six years, two employers and four clients. This is not intended to replace another functional system that works in a given setting – rather, it can serve as a starting point for such a system if none currently exists.

By "project files," we mean any files related to a project or report, such as SAS code, SAS data sets, project documentation, input and output data, or Excel files.

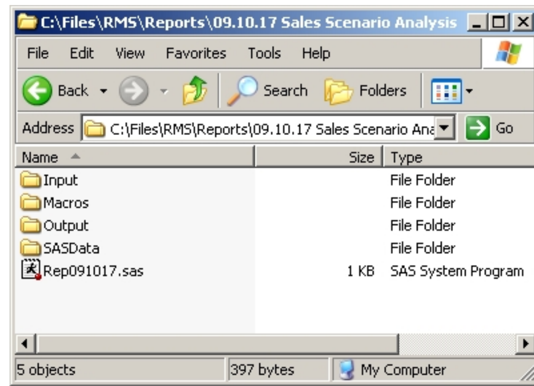
ORGANIZING FILES

The approach advocated here reflects four basic organizational ideas:

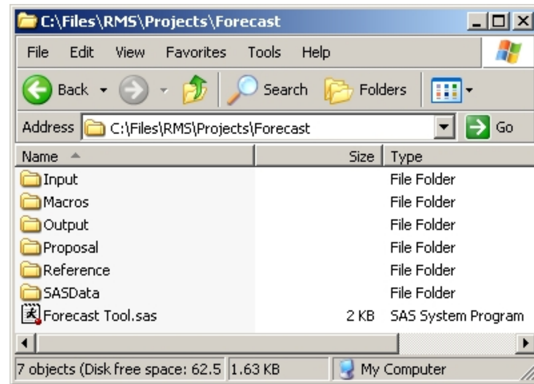
- *Never throw anything away.* A project that was canceled prematurely could be restarted later. It would be inefficient to start completely over.
- *Know where to find everything.* The report submitted six months ago that was never mentioned again could suddenly come up tomorrow, with a request that it be done over again with newer data. The code should be readily available.
- *Make the code reusable.* If the analysis is for X and the boss wants the same analysis performed for Y, why re-write the code? Instead, make the code useable for both X and Y.
- *Automate as much as possible.* Everything should be done by the program – including making new directories or making custom-formatted Excel output (see Derby (2008a,b)).



(a)



(b)



(c)

Figure 1: Examples of directory structure in general (a), for a report (b), and for a project (c).

By the proposed scheme, files should be organized

- by Company/Client
  - Projects
    - by Name
  - Reports
    - by Date Given, plus a short title or description

as shown in Figure 1(a). Generally, a *report* is something relatively minor that we are asked to do one time, whereas a *project* is something major or repeatedly updated with new data. There are no exact rules for differentiating the two, and often a report can turn into a project. As a general rule, if it has a name, it is a project. For a report, some notes are worthy of mention:

- For dates, we mean *the date given*, whether to the boss or to other coworkers. If any date pertaining to the report will be remembered by the boss, it will probably be this one (rather than, say, the date it was assigned).
- If two reports are turned in on the same day, it's not a big deal, since they will presumably have different names or descriptions. If needed, an alphabetic suffix can be added to the date (e.g., *10.07.22a*, *10.07.22b*).
- To keep the directories in chronological order, it is best to follow a date format of the year first, then the month, then the day (like *YY-MM-DD*, *YY.MM.DD*, or *YYMMDD*).

FILE ORGANIZATION

For either a project or a report, within the appropriate directory (a date and/or a name), the contents are the following:

- The *calling program*: The file `RepYYMMDD.sas` for a report or `NAME Tool.sas` for a project. This will be the only file in this directory – everything else will be in a subdirectory.
- The subdirectory *Input*: Contains the raw input data files are.
- The subdirectory *Output*: Contains the final output.
- The subdirectory *SASData*: Contains permanent SAS data sets not in the *Input* or *Output* subdirectories. Clearly this is not needed if no such permanent data sets are used.
- The subdirectory *Macros*: Contains SAS macro definitions,[1] which are accessed by the calling program `RepMMDDYY.sas`. This can have a further subdirectory, *Auxiliary* – see the **Auxiliary Macros** subsection for details.
- Various other subdirectories, such as *Reference* (for background information) or *Proposal* (for the project proposal). Generally, however, the calling program will only access files from the above four subdirectories.

Of the subdirectories listed above, only *SASData* is restricted to one data type – the others can have PDF, Excel, PowerPoint, Word, or other files. Examples of this organizational structure are shown in Figures 1(b) (for a report) and 1(c) (for a project).

CODE ORGANIZATION

For the SAS code itself, we run all parts of the report/project from the calling program, as shown in Figure 2(a)-(b). This program calls macros which group the code into specific tasks. These macros can be modified or commented out as appropriate, giving us the flexibility to easily use this same program over and over for different of tasks, giving us updated log reports each time. The calling program can be divided into a preamble and a set of macros:[2]

PREAMBLE

The preamble, as shown in Figures 2(a) and (b), includes the following:

❶ These two statements clear the log and output files before the code is run.[3] This avoids the problem of looking at the SAS log and not knowing whether the comments pertain to the most recent version of the code. Furthermore, it automates the process of clearing them, rather than doing it manually. Finally, the order in which it does so (i.e., clearing `output` before `log`) makes it such that the user can watch the log as it is being written.

❷ This is the root directory for the report or project. `LIBNAME` statements, macro definitions, and input/output directories are derived from this (as shown in Figure 3). Other roots may be included here as well, such as `exroot` for the export macro `%exportToXL` as explained in Derby (2008a). For program portability, it is best to define these in the preamble, so that they can easily be modified for use in another directory.

❸ Here we list definitions of any global macro variables used later in the program, which we expect to change frequently. If we don't change them frequently, we can put them into the `%makeSetup` macro, described below. This is often not needed in a report (Figure 2(a)).

❹ This `OPTIONS` statement tells SAS to look for macro definitions in the `&root\Macros` directory. Macros used in other directories as well (such as `&root\Macros\Auxiliary`) must be listed here as well. As shown in Figure 3, other options are listed in the `%makeSetup` macro (described below). However, these two options must be in the preamble to tell SAS to look in that macro for the other options.

❺ `%makeSetup` is a macro which serves as a repository for all setup statements, such as defining an input root, creating an output directory, or establishing a `LIBNAME`. An example is shown in Figure 3.

After this preamble, the rest of the program consists of either SAS statements or macros. A general strategy is to start with a basic set of SAS statements, but turn it into a macro as soon as it gets large. However, it can also be a good idea to group code into macros from the onset, even if only including one procedure or `DATA` statement. This keeps the code organized and maintainable from the onset.

[1] For the reader unfamiliar with macros, we can define a *macro* as simply SAS code written into a separate component, which is called by the calling program. For an introduction to macros, an internet search of "intro to SAS macros" turns up many hits. Carpenter (2004) and Burlew (2007) are good comprehensive resources.

[2] Technically `%makeSetup` should be included in the **Macros** subsection, but it serves a task that is really part of the preamble.

[3] Use `odsresults` in place of `output` to clear the listings in the Results window as well as in the Output window (assuming `listings` is an ODS output).

```
DM 'output' clear; ❶
DM 'log' clear;

%LET root = C:\Files\RMS\Reports\09.10.17 Sales Forecasts; ❷
  *where the root directory is located;

OPTIONS SASAUTOS=( "&root\Macros" ) MAUTOSOURCE; ❹


%makeSetup; ❺
  *makes the setup structures. FURTHER FUNCTIONALITY WILL NOT WORK IF THIS IS COMMENTED OUT;

*%readData;

%analyzeData;

*%exportOutput;
```

(a)

```
DM 'output' clear; ❶
DM 'log' clear;

%LET root = C:\Files\RMS\Projects\Forecast; ❷
%LET exroot = C:\SAS\ExportToXL;

%LET orig = YYC; ❸
%LET dest = YVR;
%LET datecut = 7/24/10; *cutoff date for the data;
%LET hzn = 15; *forecast horizon;

OPTIONS SASAUTOS=( "&root\Macros", "&exroot" ) MAUTOSOURCE; ❹


%makeSetup; ❺
  *makes the setup structures. FURTHER FUNCTIONALITY WILL NOT WORK IF THIS IS COMMENTED OUT;

*%readFormatData;
  *reads and segments the input data set;

*%graphData;
  *graphs the data;

%makeForecasts( fnumber=1542 );
  *produces the forecasts, cycling through various forecasting methods;

*%exportForecasts;
  *exports the forecasts onto Excel spreadsheets;

*%graphForecasts;
  *graphs the forecasts;

%put DONE!;
```

(b)

Figure 2: Calling programs for a report (a) and a project (b). For the project, we assume a project for forecasting airline demand, where we are making 15-day-out forecasts for all flights from airport YYC (Calgary = origin) to airport YVR (Vancouver, BC = destination), using data up to 7/24/10. Note that some macros are commented out.

```
%MACRO makeSetup;

  OPTIONS ORIENTATION=landscape LINESIZE=150 PAGESIZE=60 NODATE NONUMBER MCOMPILENOTE=NONE NOTES SOURCE;

  %LOCAL dir1 dir2 dir3 dir4; %*these macro variables are only used here;
  %GLOBAL outputroot enddate; %*these macro variables ar used in other macros;

  %LET outputroot = &root\Output;

  DATA _NULL_;
    dir1 = "'"||'mkdir'||' "'||"&outputroot"||'"'||"'"; %*creates directory commands to be used below;
    dir2 = "'"||'mkdir'||' "'||"&root\Data"||'"'||"'";
    CALL SYMPUTX( 'dir1', dir1 );
    CALL SYMPUTX( 'dir2', dir2 );
    CALL SYMPUTX( 'enddate', INPUT( "&datecut", mmddyy8. ) );
  RUN;

  DATA _NULL_; %*makes the directories for the graphics;
    SYSTASK COMMAND &dir1 WAIT;
    SYSTASK COMMAND &dir2 WAIT;
  RUN;

  LIBNAME rms "&root\Data";

%MEND makeSetup;
```

Figure 3: The `%makeSetup` macro for the project in Figure 2(b). Here we basically put all the code that will be used in later macros. Note that we have some `OPTIONS` here rather than in the calling program preamble. This is simply to keep the preamble smaller.

MACROS

The macro calls after the preamble are named after their respective functions, of the form `%verbNoun` for easy understanding of what is happening. These calls are generally in a necessary order (e.g., `%readFormatData` would precede `%graphData`). Furthermore, a semicolon is placed after each macro call, even though this is not necessary. This is done to allow for commenting out a given macro simply by adding an asterisk before it, as shown in Figure 2, which facilitates the common practice of running just a part of the overall code at once. Macro definitions typically have parameters (such as `%makeForecasts( fnumber=1557 )`), but they do not need to; as shown in Figure 2, we can use them purely to organize the code, and perhaps add parameters later for generalization. As shown in Figure 3, `%makeSetup` is a macro without any parameters. However, as explained below, many of these actually have hidden default parameters which call for recursion and produce results for a list, which can be applied to a specific case simply by adding a specific parameter value.

Examples of calling programs are shown in Figure 2, and an example of `%makeSetup` is in Figure 3. For further examples, see the code of the `%exportToXL` macro, as described in Derby (2008b).

Generalizing a Macro with Recursion

For flexibility and efficiency, it's useful to define a macro such that it can be used for both a general and for a specific case. As an example, let's look at the `%makeForecasts` macro in Figure 2(b). As shown here, it uses a parameter for a certain flight number:

```
%makeForecasts( fnumber=1542 );
```

In this case, forecasts would be generated for flight number 1542 only. If we wanted forecasts to be made for all flights from YYC (Calgary) to YVR (Vancouver, BC), we would have to first find all flight numbers for flights on that leg, and then loop through them:

```
%makeForecasts( fnumber=1542 );
%makeForecasts( fnumber=1543 );
%makeForecasts( fnumber=1544 );
%makeForecasts( fnumber=1545 );
%makeForecasts( fnumber=1546 );
```

However, this is tedious for two reasons: We have to list them out individually, and we have to find the flight numbers pertaining to our given origin (`&orig`) and destination (`&dest`) airports in the preamble. It would be much easier if we could just have our code automate all that by simply using the macro without parameters:

```
%makeForecasts;
```

Then we could just change the `&orig` and `&dest` values in the preamble to implement this change in `%makeForecasts` (and the other macros). However, it would be useful to *also* be able to run it individually for a specific flight number, as done for flight 1542 above. This is useful for decreasing run time for testing and drill-down purposes:

- When writing and testing the code, we can test it for one flight number before testing it for all of them at once.
- For drill-down capabilities, we can use this when looking just at a specific flight (e.g., flight 1542).

In practice, it's most useful for a combination of the above two reasons. For instance, if we are specifically interested in flight 1542, we can drill down by first looking at the results for that specific flight number, then tinker with the macro for that flight number (e.g., looking at some of the forecasting methods in detail).

This dual functionality can be accomplished by *defining a macro function recursively.* That is, we include a reference to itself within its definition. We can do this via three steps:

1. First, we define the macro for the parameter value in question:

```
%MACRO makeForecasts( fnumber );

  [Code for making forecasts]

%MEND makeForecasts;
```

2. Then we create a small macro (called an *auxiliary macro*, discussed in the next section) to determine our list of parameter values (in this case, flight numbers). There are many ways to do this – one is with `PROC SQL`:

```
%MACRO getFlightNumbers;

  PROC SQL NOPRINT;
    SELECT DISTINCT flightnumber INTO :fnumbers SEPARATED BY ' '
      FROM datasource
      WHERE orig="&orig" AND dest="&dest"
      ORDER BY by flightnumber;
  QUIT;

%MEND getFlightNumbers;
```

```
%MACRO makeForecasts( fnumber=all ); ❶

  %LOCAL i n fnumbers; ❷

  %IF &fnumber = all %THEN %DO; ❸

    %getFlightNumbers; ❹

    %LET i = 1;
    %DO %WHILE( %LENGTH( %SCAN( &fnumbers, &i ) ) > 0 ); ❺
     %LOCAL fnumber&i;
     %LET fnumber&i = %SCAN( &fnumbers, &i );
     %LET i = %EVAL( &i + 1 );
     %END;
    %LET n = %EVAL( &i - 1 );

    %DO i=1 %TO &n; ❻
     %makeForecasts( fnumber=&&fnumber&i );
     %END;

    %GOTO theend; ❼

    %END;


  [Code for making forecasts]


  %PUT fnumber=&fnumber; ❽

  %theend: ❾

%MEND makeForecasts;
```

Figure 4: Adding a recursive definition to the macro `%makeForecasts`.

3. Lastly, we change our macro definition as shown in Figure 4. Note the following:

   ❶ Adding `fnumber=all` gives us a default parameter value, `all`, that would never happen as a natural value for a flight number. As such, this can be thought of as a value which tells us to use the recursive definition.

   ❷ Following best practices, we want to keep all internal macro variables local.[4] The macro `fnumbers` is defined within the `%getFlightNumbers` macro at ❹ and accessed at ❺.

   ❸ Here is where we include the recursive definition. Code within this `%IF ... %DO` block is skipped whenever a valid (i.e., numeric) value of `fnumber` is given.

   ❹ This is the macro we defined in step 2, for finding the flight numbers.

   ❺ Here we define one macro variable, `&&fnumber&i`, for each possible flight number.[5]

   ❻ Here we loop through all possible flight numbers that we defined in our previous step, calling itself at each step.

   ❼ After we are done with the loop, we need to skip to the end of this macro, since we are essentially done.

   ❽ For testing purposes, we use a `%PUT` statement like this one to print our parameter values onto the log.

   ❾ This is the end of our macro definition.

Now running this macro without a parameter shows us in the log that this runs correctly:

Macro call		Log output
`%makeForecasts`	⇒	fnumber=1542 fnumber=1543 fnumber=1544 fnumber=1545 fnumber=1546

[4] A *local* macro variable is only defined within the macro, whereas a *global* one is defined outside of the macro.

[5] The double ampersand just tells us to resolve that after the single ampersand variables are resolved. Thus, looping through the vales of the macro variable `&i`, we are creating macro variables `&fnumber1`, `&fnumber2`, etc. For more information, see Carpenter (2004) or Burlew (2007).

Running this macro with a parameter gives us output for that flight number only:

Macro call		Log output
`%makeForecasts( fnumber=1542 )`	⇒	`fnumber=1542`

We are not limited to one parameter value for this technique; we can define a recursion with any number of parameters. Suppose that `%makeForecasts` loops through three different forecasting methods: `AddPick` (Additive Pickup), `ExSm` (Exponential Smoothing) and `ARIMA` (ARIMA). We want to design it so that it can loop through any one of them individually, as well as looping through the flight numbers individually as before. Figure 5 shows how we can modify our code in Figure 4 to also loop through the methods. This way, the statement with no parameters listed gives us all flight numbers and methods:

Macro call: `%makeForecasts` ⇒ Log output:

```
fnumber=1542, method=AddPick
fnumber=1542, method=ExSm
fnumber=1542, method=ARIMA
fnumber=1543, method=AddPick
fnumber=1543, method=ExSm
fnumber=1543, method=ARIMA
fnumber=1544, method=AddPick
fnumber=1544, method=ExSm
fnumber=1544, method=ARIMA
fnumber=1545, method=AddPick
fnumber=1545, method=ExSm
fnumber=1545, method=ARIMA
fnumber=1546, method=AddPick
fnumber=1546, method=ExSm
fnumber=1546, method=ARIMA
```

whereby running it with one parameter gives us output for that parameter value only:

Macro call: `%makeForecasts( method=ARIMA )` ⇒ Log output:

```
fnumber=1542, method=ARIMA
fnumber=1543, method=ARIMA
fnumber=1544, method=ARIMA
fnumber=1545, method=ARIMA
fnumber=1546, method=ARIMA
```

Macro call: `%makeForecasts( fnumber=1542 )` ⇒ Log output:

```
fnumber=1542, method=AddPick
fnumber=1542, method=ExSm
fnumber=1542, method=ARIMA
```

and running it with both parameters gives us output for those two parameters only:

Macro call		Log output
`%makeForecasts( fnumber=1542, method=ARIMA )`	⇒	`fnumber=1542, method=ARIMA`

```
%MACRO makeForecasts( fnumber=all, methods=all );

  %LOCAL i n fnumbers methods; ❶

  %IF &fnumber = all %THEN %DO; ❷

    %getFlightNumbers;

    %LET i = 1;
    %DO %WHILE( %LENGTH( %SCAN( &fnumbers, &i ) ) > 0 );
     %LOCAL fnumber&i;
     %LET fnumber&i = %SCAN( &fnumbers, &i );
     %LET i = %EVAL( &i + 1 );
     %END;
    %LET n = %EVAL( &i - 1 );

    %DO i=1 %TO &n;
     %makeForecasts( fnumber=&&fnumber&i, method=&method ); ❸
     %END;

    %GOTO theend;

    %END;

  %IF method = all %THEN %DO;

    %LET methods = AddPick ExSm ARIMA; ❹

    %LET i = 1;
    %DO %WHILE( %LENGTH( %SCAN( &methods, &i ) ) > 0 );
     %LOCAL method&i;
     %LET method&i = %SCAN( &methods, &i );
     %LET i = %EVAL( &i + 1 );
     %END;
    %LET n = %EVAL( &i - 1 );

    %DO i=1 %TO &n;
     %makeForecasts( fnumber=&fnumber, method=&&method&i ); ❺
     %END;

    %GOTO theend;

    %END;


  [Code for making forecasts]


  %PUT fnumber=&fnumber, method=&method;

  %theend:

%MEND makeForecasts;
```

Figure 5: Adding a two-parameter recursive definition to the macro `%makeForecasts`.

A few things to note about the code in Figure 5:

❶ We need to add a new local macro variable for our new list of parameter values (`methods`).

❷ It doesn't matter what order we put the parameters. Here the flight numbers are the first level and the methods are the second, giving us the order shown in the log output shown on the preceding page.

❸ When calling the macro, keep track of which parameters are being looped. Here, we are looping through the flight numbers (`&&fnumber&i`) and keeping the method constant (`&method`). This is very different from the call at ❺.

❹ For simplicity, we are listing the forecasting methods individually in a macro list (which couldn't be done with the flight numbers because that was dependent on the data). If this list is accessed from multiple macros, it's best for code maintenance to put this into a separate macro, which could be called `%getForecastMethods`.

❺ As opposed to ❸, we are now looping through the methods (`&&method&i`) and keeping the flight number constant (`&fnumber`).

Auxiliary Macros

We can think of an *auxiliary macro* as a macro that is used in other macros. Such a macro is usually short, and may or may not be general. For example, it might just be one line long such as in defining a list of forecasting methods:

```
%MACRO getForecastMethods;

  %LET methods = AddPick ExSm ARIMA;

%MEND getForecastMethods;
```

This way, if a forecasting method is added later on, changing that line is all that is necessary, no matter how many other macros access it. However, such a line may be better used in the calling program preamble, depending on how often it gets changed. Other auxiliary macros are more generalized, such as the following, which adds a character string to every variable name in a given data set:

```
%MACRO renameVars( inputdata, outputdata, char );

  %LOCAL vars i;

  PROC CONTENTS DATA=&inputdata OUT=meta NOPRINT;
  RUN;

  PROC SQL NOPRINT;
    SELECT name
      INTO :vars SEPARATED BY ' '
    FROM meta;
  QUIT;

  DATA &outputdata;
    SET &inputdata;
    %LET i = 1;
    %DO %WHILE( %LENGTH( %SCAN( &vars, &i ) ) > 0 );
        rename %SCAN( &vars, &i ) = %SCAN( &vars, &i )&char;
        %LET i = %EVAL( &i + 1 );
        %END;
    RUN;

%MEND renameVars;
```

These are just examples of what can be considered auxiliary macros. Regardless of how generalized or specific they are, for organization, they should be included in a subdirectory of the *Macros* subdirectory, named *Auxiliary*. A rule of thumb: If a macro is not called from the calling program, it should be in the *Auxiliary* subdirectory.

CONCLUSIONS

This is not meant to be a comprehensive document about file organization. Rather, it is just a suggestion for one organizational structure which has worked for a variety of settings over a few years. I hope it can be helpful.

REFERENCES

Burlew, M. (2007), *SAS Macro Programming Made Easy*, second edn, SAS Institute, Inc., Cary, NC.

Carpenter, A. (1999), Getting more for less: A few SAS programming efficiency issues, *Proceedings of the Twelfth Northeast SAS Users Group Conference*.
http://www.nesug.org/Proceedings/nesug99/cc/cc199.pdf

Carpenter, A. (2004), *Carpenter's Complete Guide to the SAS Macro Language*, second edn, SAS Institute, Inc., Cary, NC.

Chidambaram, K. (2005), Speed up your SAS programs, *Proceedings of the 2005 Pharmaceutical Industry SAS Users Conference*, paper PO04.
http://www.lexjansen.com/pharmasug/2005/posters/po04.pdf

Derby, N. (2008a), Revisiting DDE: An updated macro for exporting SAS data into custom-formatted Excel spreadsheets, Part I – Usage and examples, *Proceedings of the 2008 SAS Global Forum*, paper 259-2008.
http://www2.sas.com/proceedings/forum2008/259-2008.pdf

Derby, N. (2008b), Revisiting DDE: An updated macro for exporting SAS data into custom-formatted Excel spreadsheets, Part II – Programming details, *Proceedings of the 2008 SAS Global Forum*, paper 260-2008.
`http://www2.sas.com/proceedings/forum2008/260-2008.pdf`

Dosani, F., Fecht, M. and Eckler, L. (2010), Creating easily-reusable and extensible processes: Code that thinks for itself, *Proceedings of the 2010 SAS Global Forum*, paper 004-2010.
`http://support.sas.com/resources/papers/proceedings10/004-2010.pdf`

Fecht, M. and Stewart, L. (2008), Are your SAS programs running you?, *Proceedings of the 2008 SAS Global Forum*, paper 164-2008.
`http://www2.sas.com/proceedings/forum2008/164-2008.pdf`

Jolley, L. and Stroupe, J. (2007), Dear Miss SASAnswers: A guide to SAS efficiency, *Proceedings of the 2007 SAS Global Forum*, paper 042-2007.
`http://www2.sas.com/proceedings/forum2007/042-2007.pdf`

Lafler, K. P. (2000), Efficient SAS programming techniques, *Proceedings of the Twenty-Fifth SAS Users Group International Conference*, paper 146-25.
`http://www2.sas.com/proceedings/sugi25/25/hands/25p146.pdf`

Langston, R. (2005), Efficiency considerations using the SAS system, *Proceedings of the Thirtieth SAS Users Group International Conference*, paper 002-30.
`http://www2.sas.com/proceedings/sugi30/002-30.pdf`

Winn Jr., T. J. (2004), Guidelines for coding of SAS programs, *Proceedings of the Twenty-Ninth SAS Users Group International Conference*, paper 258-29.
`http://www2.sas.com/proceedings/sugi29/258-29.pdf`

ACKNOWLEDGMENTS

I thank my present and past employers and clients for giving me such a high degree of flexibility in organizing my projects. I very much thank SAS technical support for helping me learn the basics of macro programming. I also thank Ron Fehd for providing me with a LaTeX template for SAS conference papers (used here).

Lastly, and most importantly, I thank Charles for his patience and support.

SAS and all other SAS Institute Inc. product or service names are registered trademarks or trademarks of SAS Institute Inc. in the USA and other countries. ® indicates USA registration.

Other brand and product names are trademarks of their respective companies.

Copyright 2010 by Nate Derby