The SAS System in the Pharmaceutical Industry

David Shannon, Amadeus Software Limited

ABSTRACT

This paper discusses the uses of the SAS System within the pharmaceutical industry. It offers suggestions for programming best practices and coding hints and tips along the way for young Statistician's and Statistical Programmers.

Consideration will be given to the organisation of programs for efficient storage and re-use.

Finally an overview will presented of Enterprise Guide, a new environment for SAS programming. A brief discussion will be made on its use in the pharmaceutical industry today.

This paper is of benefit to programmers and statisticians of the pharmaceutical industry who use the SAS System for data derivation and the production of tables and figures.

INTRODUCTION

The pharmaceutical industry is the most highly regulated industry that I have worked in, where the regulations and guidelines directly impact the work of SAS Programmers, Statistician's and Data Managers who use the SAS System.

This paper attempts to draw together some of the best practices which I have observed. The reader should note that practices within their own organisation may are quite probably already well defined.

Whilst levels of expenditure on infrastructure, software applications and the education of standard practices are very high amongst global pharmaceutical organisations, the same purchasing power is not always available to localised organisations. With this comment in mind this paper sets forth to explore practices, examples, tips and techniques which may be employed by all.

PROGRAMMING IN THE PHARMACEUTICAL INDUSTRY

PROGRAMMING IN LAYERS

Organising your SAS programs within a clinical study, into tiers (or logical layers) is the first step to avoiding duplication of code and effort.

The first layer (i.e. group) of programs are organised together to extract the required clinical data from the source to which it is provided to us and perform any normalisation and variable attribute assignments. Tasks such as joining tables to gather like fields are performed, along with derivation of end-points or other variables, as required. Data are not aggregated when stored after this derivation. This allows quality control procedures to easily verify transformed data against the source system database.

There are several terms used between organisations for this task, such as deriving, value adding etc. Extract, Transform and Load (ETL) is another term used across many industries for this process.

The middle layer is where data may be summarised and/or analysed, ultimately for reporting. Copies of summarised data are often held for both quality control purposes and ease of reporting. The data within these tables often holds real value in answering the questions set by the trial being analysed.

Note that it is highly unlikely that a well designed set of SAS programs for any given clinical study will have single programs generating single outputs. The figure below does not intend to depict this! It is quite likely that one macro, called several times with various parameters will generate several sets of outputs, be they tables or reports.

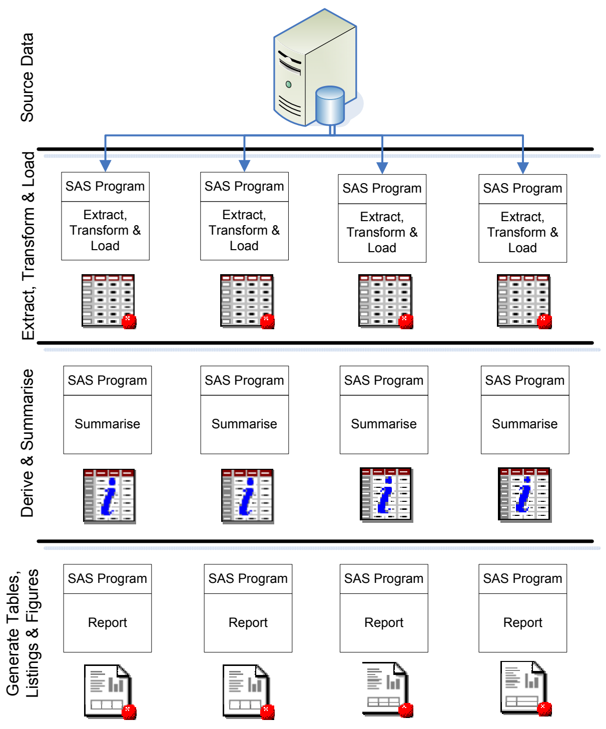

Source Data
Extract, Transform & Load
SAS Program
Extract, Transform & Load
Derive & Summarise
SAS Program
Summarise
Generate Tables, Listings & Figures
SAS Program
Report

Figure 1: Logical Layers in SAS Programming

Although very tempting to begin programming immediately when given access to data and an analysis plan, the key to arriving at a well organised set of programs and macros etc. is planning. Examining the commonality and structure of the data captured with respect to the requirements of the tables, listings, figures and any additional ad-hoc work requested can help greatly reduce the amount of programming and duplicated data storage required. This is a task that really does benefit from the input of an experienced programmer.

Associated with the planning of your programming is the testing. Assuring accurately written programmes is critical in ensuring the validity of results and conclusions drawn.

There are various methods of validating results, including independent dual programming, tracing the path of a subset of subjects through the data, or code reviews. Most commonly independent programming is performed outside of organisations that use highly validated generic SAS programs which are reused between studies.

WRITING CLEAR PROGRAMS

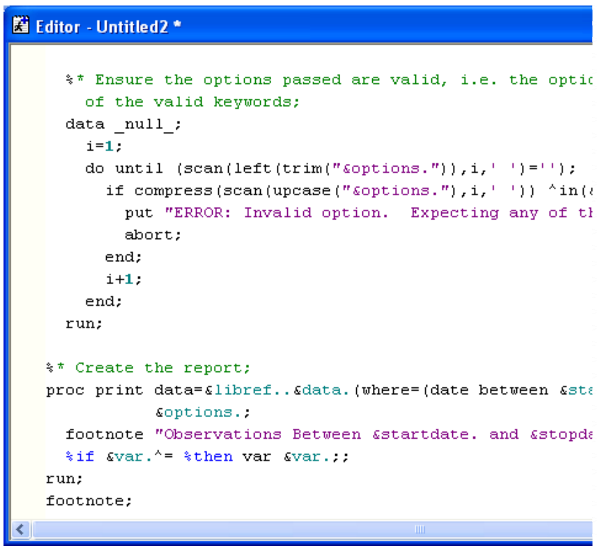
Within your team, department or even organisation wide it is recommended that a standard programming style is adopted.

The benefit is consistency between programmers. This decreases the complexity and effort associated of individual users working with code written by other members of the team.

The author has observed some quite elaborate coding conventions which request code to be written with certain key words in uppercase, camel case or lowercase.

There is little benefit, or indeed it is a cost, from requesting programmers spend additional effort in enforcing complex coding rules.

Simplicity is the key to encouraging flowing and effective programming. Consider the following program written in the style adopted by many organisations (including that of Amadeus):



```
Editor - Untitled2 *

   %* Ensure the options passed are valid, i.e. the optic
     of the valid keywords;
   data _null_;
     i=1;
     do until (scan(left(trim("&options.")),i,' ')='');
       if compress(scan(upcase("&options."),i,' ')) ^in(
         put "ERROR: Invalid option.  Expecting any of th
         abort;
       end;
       i+1;
     end;
   run;

 %* Create the report;
 proc print data=&libref..&data.(where=(date between &sta
            &options.;
   footnote "Observations Between &startdate. and &stopda
   %if &var.^= %then var &var.;;
 run;
 footnote;
```

Figure 2: Coding Conventions

The above figure demonstrates a data step, proc step and global statement from within a macro. All SAS code is written in lowercase for both speed and ease of writing. Only strings of text are case sensitive in SAS language.

Comments are written as in-line comments. Block comments cannot exist within other block comments without causing syntax errors. Hence in-line comments are used whenever possible (in this above figure a percent symbol prefixes the asterisk to hide the comment from the macro processor).

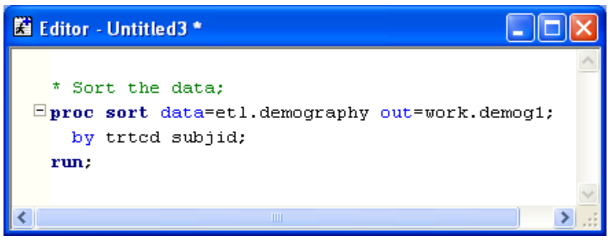
Indenting is made between data step boundaries and DO blocks. At Amadeus two spaces are used to indent the code whilst tabs are also commonly used.

Programs should always be clearly documented. All too often this is the area of SAS programming which suffers when the author is up against pressurised deadlines. It is only to the detriment of the author, their colleagues, or even the organisation when the code is opened and information regarding the programs activities cannot be easily obtained.

Every program should include a header which considers the following items:

- Program or Macro name
- A description of its function
- Any required or options parameters to macros, including a description of their usage and list of values where constrained
- Example of usage
- Any required input data sources, files etc. to the program
- Any outputs generated by the program
- An example of usage
- Contact information of the person/department who maintains the code
- Change control history

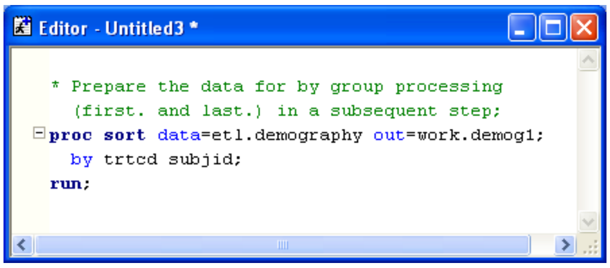
A further component of program documentation is the commenting of code. Code should be commented to explain the reasons of the code. Consider the following:



```
Editor - Untitled3 *

 * Sort the data;
proc sort data=etl.demography out=work.demog1;
  by trtcd subjid;
run;
```

Figure 3: Poor Commenting

The figure above demonstrates the worst, yet frequently observed, type of comment. It states the obvious, but is missing the point: Why did the original programmer sort the data?



```
Editor - Untitled3 *

 * Prepare the data for by group processing
   (first. and last.) in a subsequent step;
proc sort data=etl.demography out=work.demog1;
  by trtcd subjid;
run;
```

Figure 4: Better Commenting

The above figure represents a better comment. It describes why the data are sorted and will lead any future reader to a subsequent step in the program.

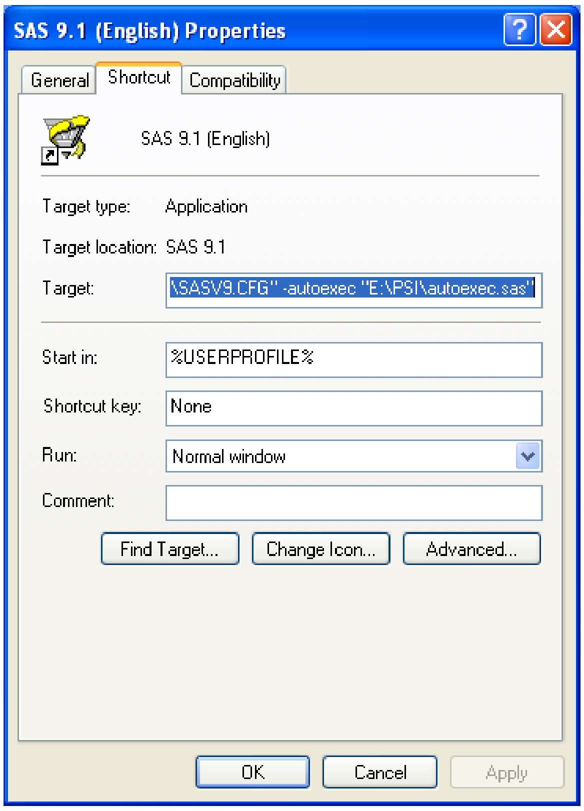
CONFIGURING THE SAS ENVIRONMENT

There are numerous system options that can be set on any SAS session, far too many for full discussion here. The single most useful option is the designation of a SAS program that runs each time your SAS session start, i.e. an AUTOEXEC program.

The typical uses of an autoexec program are to assign libraries, global SAS system options, such as macro debugging options, locations of a format catalogs etc. The benefits are such that a single autoexec program can be created which is shared by several people enabling access to data storage and SAS sessions to be consistent and transparent between groups of users. Consider autoexec programs for settings options such as:

- LIBNAME assignments
- FILENAME assignments
- Titles
- Footnotes
- Global system options such as
  - Macro autocall libraries
  - Macro debugging
  - Format catalog search paths
  - Paper and margin settings
  - Message level information
  - Output date and time settings

A common way of associating an AUTOEXEC program with a SAS session is to create a new shortcut icon to your SAS session and append the –autoexec option onto the target of the shortcut, as shown in the figure below.



SAS 9.1 (English)

Target type: Application

Target location: SAS 9.1

Target: \SASV9.CFG" -autoexec "E:\PSI\autoexec.sas"

Start in: %USERPROFILE%

Shortcut key: None

Run: Normal window

Comment:

Figure 5: Associating an Autoexec with a SAS Session

CODE ORGANISATION

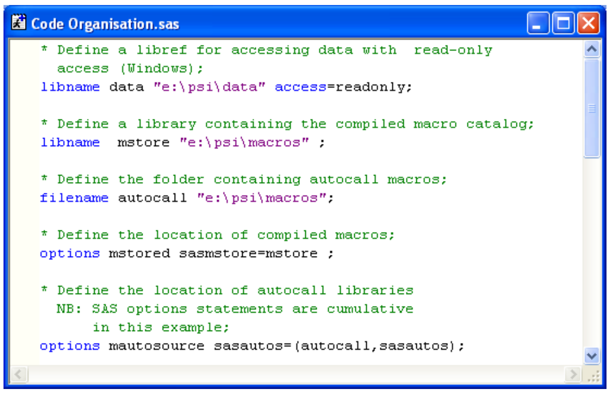
Granting read-only access to data sources is recommended to prevent accidental deletion or manipulation of data. Data sources are typically read-only when the source is a relational database; however this can also be controlled via the ACCESS=REAONLY option on a LIBNAME statement (Windows).

Use of the macro language is very common in every industry that uses the SAS. Macro brings a high level of flexibility and the ability to repetitively use code with the minimum of effort.

It is good practice to isolate repetitively used macros from the programs which call those macros. Defining a macro library, or libraries, for a SAS session allows seamless access to those macros.

There are two mechanisms of storing macros; in an autocall library where each macro is stored in a .SAS program file of the same name as the macro. The second method stores compiled macros in a SAS catalog. Typically one or the other is used, but technically it is possible to access both styles simultaneously.

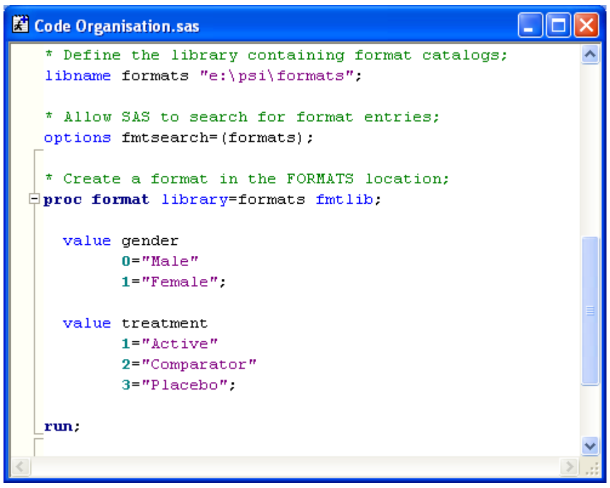
Figure 6 shows SAS statements used to access macro libraries in each of autocall and compiled catalogs.



```
* Define a libref for accessing data with  read-only
  access (Windows);
libname data "e:\psi\data" access=readonly;

* Define a library containing the compiled macro catalog;
libname  mstore "e:\psi\macros" ;

* Define the folder containing autocall macros;
filename autocall "e:\psi\macros";

* Define the location of compiled macros;
options mstored sasmstore=mstore ;

* Define the location of autocall libraries
  NB: SAS options statements are cumulative
      in this example;
options mautosource sasautos=(autocall,sasautos);
```

Figure 6: Macro Libraries

Formats are a commonly used method of storing the true meaning for coded information captured in databases and SAS data sets. Format catalog search paths can be defined within your SAS session as shown in the figure below.



```
* Define the library containing format catalogs;
libname formats "e:\psi\formats";

* Allow SAS to search for format entries;
options fmtsearch=(formats);

* Create a format in the FORMATS location;
proc format library=formats fmtlib;

  value gender
        0="Male"
        1="Female";

  value treatment
        1="Active"
        2="Comparator"
        3="Placebo";

run;
```

Figure 7: Format Libraries

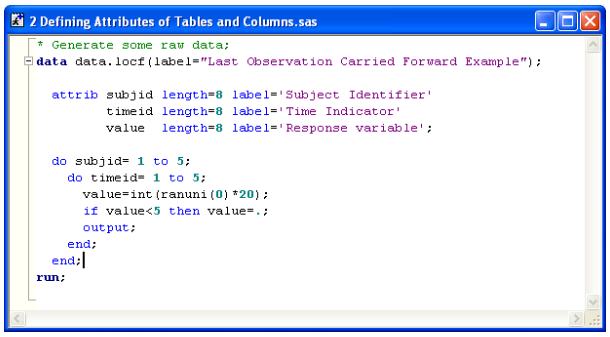
TIPS AND TRICKS

Defining Attributes of Tables and Columns

It is good practice to ensure that all columns stored in output data sets are labelled, the length is appropriate and any format are permanently associated with the variable.

Lengths determine the amount of data held in the column, this is particularly relevant to character variables. Ensure that character variables are long enough to contain the data required to be stored, but defining variables with excessively long widths causes unnecessary disk space to be used and SAS extra processing effort when operating on those variables.

The figure below shows an ATTRIB statement being used to define the attributes of several variables concurrently.



```
* Generate some raw data;
data data.locf(label="Last Observation Carried Forward Example");

  attrib subjid length=8 label='Subject Identifier'
         timeid length=8 label='Time Indicator'
         value  length=8 label='Response variable';

  do subjid= 1 to 5;
    do timeid= 1 to 5;
      value=int(ranuni(0)*20);
      if value<5 then value=.;
      output;
    end;
  end;
run;
```

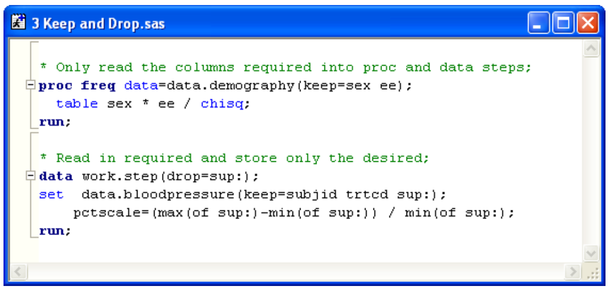
Figure 8: Defining Attributes

Also note that a label is added to the data set which is permanently stored. This is good practice because it allows metadata describing utilities, such as Proc CONTENTS, and other SAS 9 utilities to display meaningful information about the contents of the table.

Efficiencies through Keep and Drop

Selecting only the variables that are needed to be kept for the current proc or data step can decrease the amount of processing work required by SAS.

The following figure demonstrates both a procedure and a data step where keep and drop are used as data step options.



```

* Only read the columns required into proc and data steps;
proc freq data=data.demography(keep=sex ee);
  table sex * ee / chisq;
run;

* Read in required and store only the desired;
data work.step(drop=sup:);
set  data.bloodpressure(keep=subjid trtcd sup:);
    pctscale=(max(of sup:)-min(of sup:)) / min(of sup:);
run;
```

Figure 9: Use of Keep and Drop

Remember that if a keep (or drop) data set option is used when reading data into a step (which incidentally is the most efficient means of selecting variables to be kept and operated on) then those, and only those, variables will be available to the data step. Hence if a variable is required during the data step for performing calculations, but not required to be stored in the output data step, then it should be kept in the set (merge, update or modify) statement and dropped on the data statement.

Where versus If

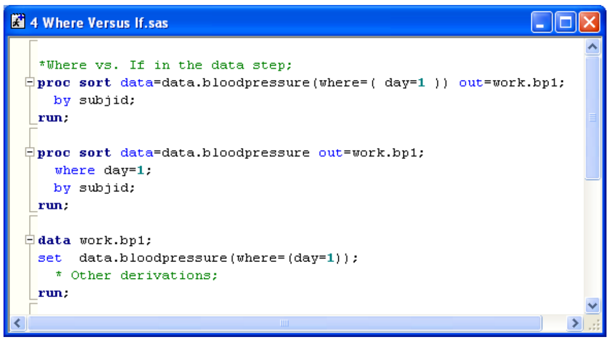
Filtering data in the SAS system can be performed by both the WHERE and IF statements in the data step. WHERE statements can also be used to filter data within proc steps.

As a data set option a where clause takes the following form:

```
<libref.>table (WHERE = (clause))
```

The infrequent programmer is often deterred by the additional parentheses required for this style, however the where statement can also be used as a statement on its own in both proc and data steps.

The figure below demonstrates each of these uses:



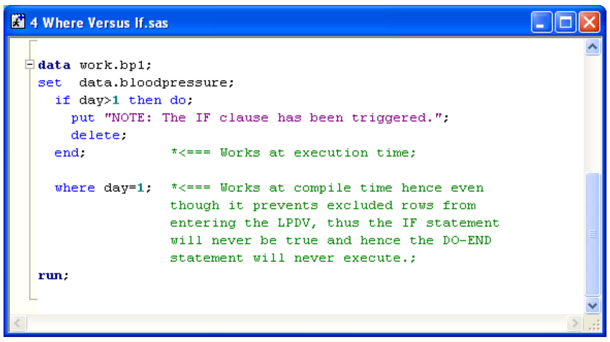
```

*Where vs. If in the data step;
proc sort data=data.bloodpressure(where=( day=1 )) out=work.bp1;
  by subjid;
run;

proc sort data=data.bloodpressure out=work.bp1;
  where day=1;
  by subjid;
run;

data work.bp1;
set  data.bloodpressure(where=(day=1));
  * Other derivations;
run;
```

Figure 10: Filtering Data with Where Clauses

Each of the first two proc steps perform the same task i.e. filtering the input table to select only the rows where DAY=1. Note that an OUT= option is used in both cases; if not the input data set is overwritten causing all rows but those selected to be deleted! A where clause can be used as a data set option whenever a data set is called, hence within a data step also.

The WHERE statement is preferred for efficiency over the IF clause as rows excluded by the clause never enter the logical program data vector (LPDV).

This is the case because of the special behaviour of the WHERE statement in that it is effectively executed before any other code in the proc or data step, i.e. at compile time. The figure below demonstrates this. One would expect that the IF statement is evaluated as true and excludes rows before the WHERE clause is reached.



```

data work.bp1;
set  data.bloodpressure;
  if day>1 then do;
    put "NOTE: The IF clause has been triggered.";
    delete;
  end;             *<=== Works at execution time;

  where day=1;     *<=== Works at compile time hence even
                   though it prevents excluded rows from
                   entering the LPDV, thus the IF statement
                   will never be true and hence the DO-END
                   statement will never execute.;
run;
```

Figure 11: Where versus If

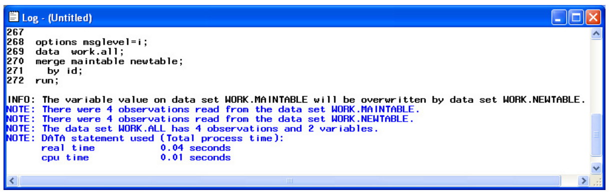
As discussed above the WHERE statement prevents any rows where DAY is not equal to 1 from entering the data step, hence the IF DO – END block is never executed.

MSGLEVEL = I

An extra level of information can be added to the SAS log via the MSGLEVEL system option.

Although this option provides extra information via INFO: messages for various aspects of SAS programming, such as the use of indexes on tables, it can also provide particularly useful information when merging tables.

Consider the situation where two tables are to be match merged, where several variables exist on each table. It is sometimes observed that one or more variables exist with a common name but contain different data. If these variables are not correctly handled during the merge the data from the right most table on the merge statement, will overwrite the data from the first table on the merge statement. This may result in erroneous use of the data.
The following figure shows a log file where the INFO statement warns of such an event.



```
Log - (Untitled)
267
268  options msglevel=i;
269  data  work.all;
270  merge maintable newtable;
271    by id;
272  run;

INFO: The variable value on data set WORK.MAINTABLE will be overwritten by data set WORK.NEWTABLE.
NOTE: There were 4 observations read from the data set WORK.MAINTABLE.
NOTE: There were 4 observations read from the data set WORK.NEWTABLE.
NOTE: The data set WORK.ALL has 4 observations and 2 variables.
NOTE: DATA statement used (Total process time):
      real time           0.04 seconds
      cpu time            0.01 seconds
```

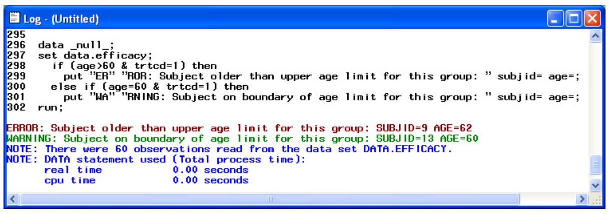
Figure 12: MSGLEVEL=I

By default when a SAS session is invoked the level of MSGLEVEL is N.

Message in a Log File

Frequently programmers add messages into the SAS log to indicate status of data steps, macros and even the values of data.

It can be useful to use the default keywords to indicate notes, warnings and errors as the log window in SAS DMS will highlight these messages as shown in the figure below:



```
Log - (Untitled)
295
296  data _null_;
297  set data.efficacy;
298    if (age>60 & trtcd=1) then
299      put "ER" "ROR: Subject older than upper age limit for this group: " subjid= age=;
300     else if (age=60 & trtcd=1) then
301      put "WA" "RNING: Subject on boundary of age limit for this group: " subjid= age=;
302  run;

ERROR: Subject older than upper age limit for this group: SUBJID=9 AGE=62
WARNING: Subject on boundary of age limit for this group: SUBJID=13 AGE=60
NOTE: There were 60 observations read from the data set DATA.EFFICACY.
NOTE: DATA statement used (Total process time):
      real time           0.00 seconds
      cpu time            0.00 seconds
```

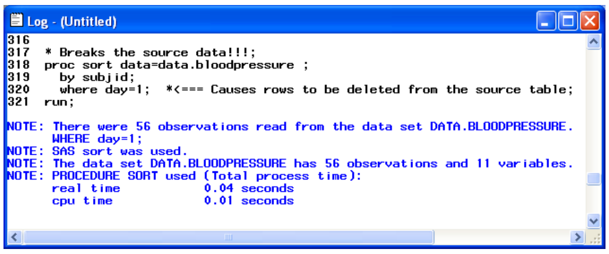
Figure 13: Message in a Log File

The code used breaks up the keywords within the PUT statements. This allows a reader of the log file to search for the text "ERROR" without yielding a search result which is part of the code itself. Only true error messages generated by the SAS System will be located.

Don't Overwrite the Input Table

To enable the traceability of data through a program it is good practice to always create a new output data set as a result of a proc or data step.

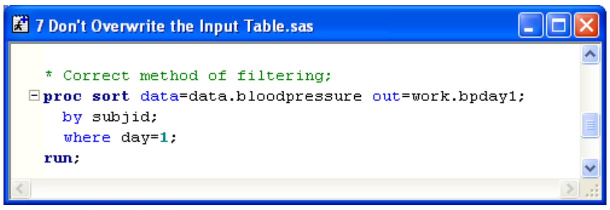
Getting into the habit of this practice can also help the accidental deletion of data. Consider the following log file where a proc sort is used with a WHERE statement. Observations are filtered and only 56 (of the 370) are selected, however as no output data set is specified the default action of the SORT procedure is to overwrite the input causing 314 observations to be deleted from the source table!



```
Log - (Untitled)
316
317  * Breaks the source data!!!;
318  proc sort data=data.bloodpressure ;
319    by subjid;
320    where day=1;  *<=== Causes rows to be deleted from the source table;
321  run;

NOTE: There were 56 observations read from the data set DATA.BLOODPRESSURE.
      WHERE day=1;
NOTE: SAS sort was used.
NOTE: The data set DATA.BLOODPRESSURE has 56 observations and 11 variables.
NOTE: PROCEDURE SORT used (Total process time):
      real time           0.04 seconds
      cpu time            0.01 seconds
```

Figure 14: Overwriting Source Data

The recommended practice is to always create an output data set with a new name, in the WORK library if permanent storage is not required.



```
* Correct method of filtering;
proc sort data=data.bloodpressure out=work.bpday1;
  by subjid;
  where day=1;
run;
```

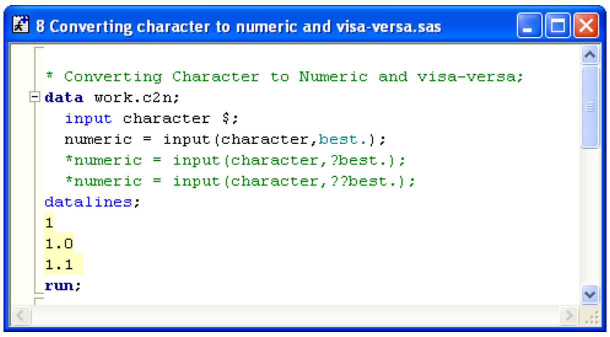
Figure 15: Creating a New Output Data Set

Converting Character Variables to Numeric and Visa-Versa

Causing implicit conversions of character data to numeric and visa-versa leaves NOTE messages in the log file which indicate that SAS has made the conversion and in certain circumstances could lead to data not being converted as one might expect.

It is recommended that the INPUT and PUT functions are used to convert between data types.

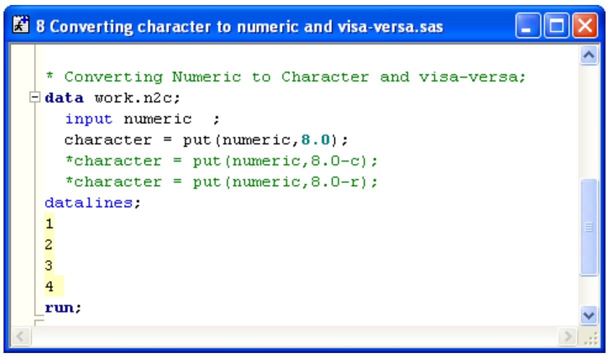
Figures 16 and 17 demonstrate the use of these functions for character to numeric and numeric to character conversions, respectively.



```
* Converting Character to Numeric and visa-versa;
data work.c2n;
  input character $;
  numeric = input(character,best.);
  *numeric = input(character,?best.);
  *numeric = input(character,??best.);
datalines;
1
1.0
1.1
run;
```

Figure 16: Converting Character Variables to Numeric

The example shown above has two commented functions that show the optional use of a single or double question mark prefixing the target format. Should data be contained in the character variable which cannot be converted to numeric (such as a letter) the default behaviour (without question marks) is to generate an invalid data message, print the contents of the LPDV to the log file and set the internal data step variable _ERROR_ to 1 (i.e. true) for the current observation. A single question mark will prevent the message and LPDV being printed to the log and a double question mark will force the _ERROR_ to remain 0 (i.e. false). This can be useful when the consequences of losing data during the conversion are understood, or more appropriately data cleansing routines and validation programs highlight such data.



```
* Converting Numeric to Character and visa-versa;
data work.n2c;
  input numeric  ;
  character = put(numeric,8.0);
  *character = put(numeric,8.0-c);
  *character = put(numeric,8.0-r);
datalines;
1
2
3
4
run;
```

Figure 17: Converting Numeric Variables to Character

The above figure demonstrates the conversion of numeric to character values which contains two commented uses of the PUT function with additional format modifiers. The –L, -C and –R force the character variable to be left, centre and right aligned with the character column respectively.

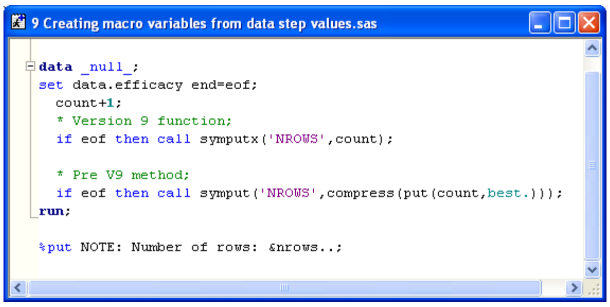
Creating Macro Variables from Data Step Values

Storing data in a macro variable, either derived within a data step or directly from a data set is a common programming task.

The most efficient way of creating a macro variable (or variables) from a data step is shown in Figure 18. A variable (count) increments on each iteration of a data step before its value is stored in a macro variable on the final observation read from the input data set.

The Call SYMPUT function is used to create and assign the value to the macro variable. Beware that if a numeric variable is to be assigned into the macro variable then the appropriate conversion from numeric to character should be performed. As a result the value should be left aligned and trimmed to avoid storing additional blank space in the macro variable.

The example which follows uses the compress function to remove all spaces from the result of the PUT function.



```
data _null_;
set data.efficacy end=eof;
  count+1;
  * Version 9 function;
  if eof then call symputx('NROWS',count);

  * Pre V9 method;
  if eof then call symput('NROWS',compress(put(count,best.)));
run;

%put NOTE: Number of rows: &nrows..;
```

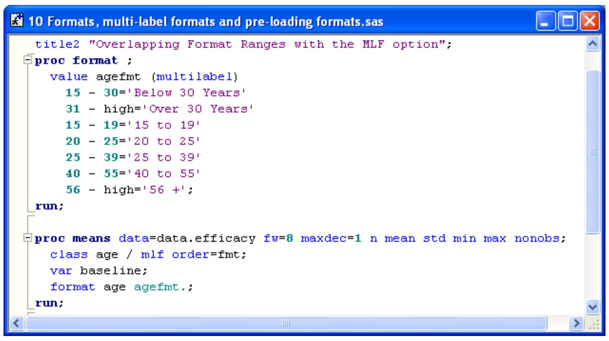
Figure 18: Creating Macro Variables from the Data Step

SAS 9 introduces a new Call function, Call SYMPUTX which performs the conversion from numeric to character before left aligning and trimming the result automatically.

Formats, Multi-Label Formats and Pre-Loading Formats

A useful feature of formats is demonstrated below whereby data can be grouped together in multiple groups during a single call to a procedure for summarising.

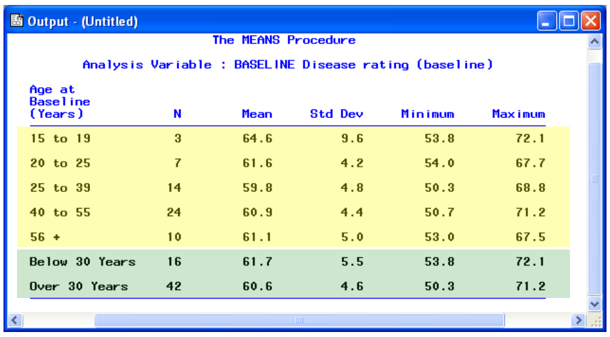
The sample below calls on the MULTILABEL option which then allows overlapping ranges of data to be specified in the format definition.



```
 title2 "Overlapping Format Ranges with the MLF option";
proc format ;
  value agefmt (multilabel)
    15 - 30='Below 30 Years'
    31 - high='Over 30 Years'
    15 - 19='15 to 19'
    20 - 25='20 to 25'
    25 - 39='25 to 39'
    40 - 55='40 to 55'
    56 - high='56 +';
run;

proc means data=data.efficacy fw=8 maxdec=1 n mean std min max nonobs;
  class age / mlf order=fmt;
  var baseline;
  format age agefmt.;
run;
```

Figure 19: Multi-Label Formats

A call to Proc MEANS (also SUMMARY, TABULATE and REPORT facilitate this) with the MLF option on the CLASS statement requests that multiple ranges are summarised as the highlighted results in Figure 20 show:

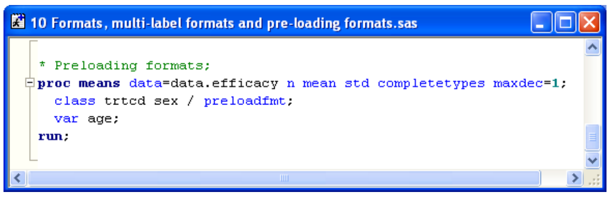


```
                          The MEANS Procedure

        Analysis Variable : BASELINE Disease rating (baseline)

 Age at
 Baseline
 (Years)           N        Mean     Std Dev     Minimum     Maximum
 15 to 19          3        64.6         9.6        53.8        72.1
 20 to 25          7        61.6         4.2        54.0        67.7
 25 to 39         14        59.8         4.8        50.3        68.8
 40 to 55         24        60.9         4.4        50.7        71.2
 56 +             10        61.1         5.0        53.0        67.5
 Below 30 Years   16        61.7         5.5        53.8        72.1
 Over 30 Years    42        60.6         4.6        50.3        71.2
```

Figure 20: Results with Multi-Label Formats

Preloading formats can be a useful way of summarising data that isn't there!

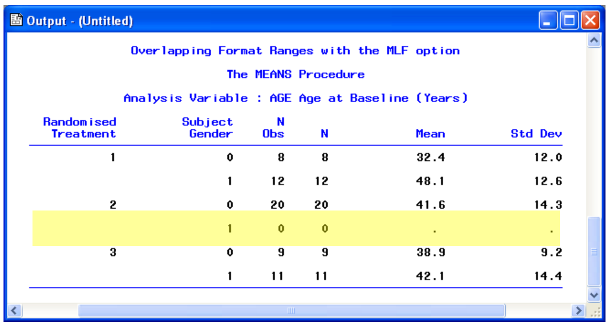
Supposing, as in the data shown below, there are three treatment groups 1, 2 and 3 where group 2 has no female subjects (code 1). Ordinarily no row would be shown for this subset as there are no data summarise.



```
 * Preloading formats;
proc means data=data.efficacy n mean std completetypes maxdec=1;
  class trtcd sex / preloadfmt;
  var age;
run;
```

Figure 21: Preloading Formats

By use of the COMPLETETYPES option on the proc statement and PRELOADFMT option on the CLASS statement the procedure summarises all combinations of class variables with or without observations in the source data.



```
             Overlapping Format Ranges with the MLF option

                         The MEANS Procedure

             Analysis Variable : AGE Age at Baseline (Years)

 Randomised          Subject    N
  Treatment           Gender  Obs     N            Mean        Std Dev
 ---------------------------------------------------------------------
          1                0    8     8            32.4           12.0

                           1   12    12            48.1           12.6

          2                0   20    20            41.6           14.3

                           1    0     0               .              .

          3                0    9     9            38.9            9.2

                           1   11    11            42.1           14.4
 ---------------------------------------------------------------------
```

Figure 22: Results of Preloading Formats

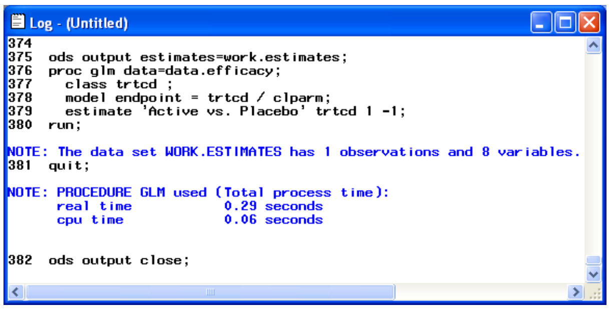
The effect of these options is to produce the additional row of output highlighted above.

ODS OUTPUT

The output delivery system is mostly associated with its ability to create reports for numerous formats. An ability exists also to create output data sets from virtually any part of any procedures output, even when the traditional output statement, or OUT= option, from a given procedure does not facilitate the capture of all results generated.

A typical example is the use of the ESTIMATE statement in the GLM procedure. Used to calculate differences between adjusted means, the associated confidence interval and p-value, the GLM procedure does not capture this output directly into a SAS data set.

As the following log file shows, using ODS OUTPUT and the appropriate object name (i.e. ESTIMATES in this example) allows a data set to be generated containing these results.



```
374
375  ods output estimates=work.estimates;
376  proc glm data=data.efficacy;
377    class trtcd ;
378    model endpoint = trtcd / clparm;
379    estimate 'Active vs. Placebo' trtcd 1 -1;
380  run;

NOTE: The data set WORK.ESTIMATES has 1 observations and 8 variables.
381  quit;

NOTE: PROCEDURE GLM used (Total process time):
      real time           0.29 seconds
      cpu time            0.06 seconds


382  ods output close;
```

Figure 23: ODS OUTPUT

Object names can be determined by submitting ODS TRACE ON; before a procedure call. Object names are printed in the log file.

It's a good idea to submit ODS TRACE OFF; once the object names are determined as the amount of extra information generated in the log file quickly becomes overwhelming!

Visualising Data

The SAS System has an immensely flexible programming language for generating graphics through SAS/Graph, the annotate facility and the graphics stream interface.

It has been the author's experience that graphics with the SAS System are frequently overlooked because of the perceived effort involved in learning the appropriate syntax to produce the graphics required. This concerns me, as one of the best ways of understanding the data we are working with is to visualise it. After all I'm sure we are all familiar with the phrase "A picture paints a thousand words…". Before I digress further (I'm sure the topic is worthy of another paper!) let me return to the hints and tips theme of this section:

The SAS System has made great strides in recent versions to simplify the code required to get professional quality graphics of data. Consider Figure 24 below.

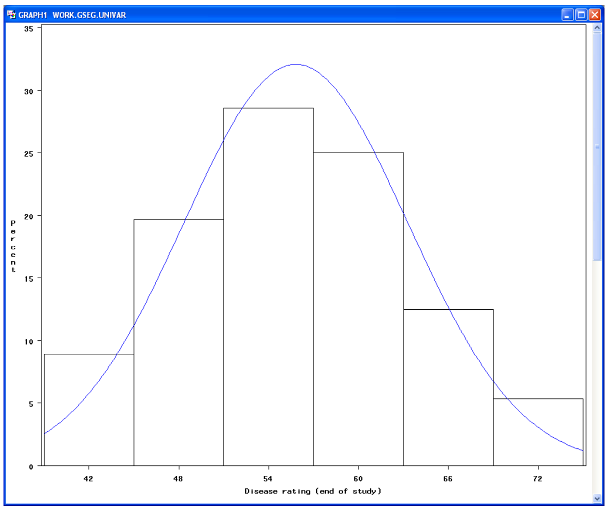
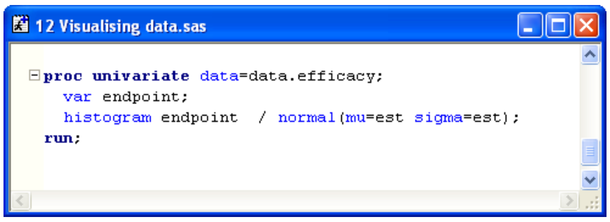


Figure 24: Histogram from Proc Univariate

Just four lines of code were used to comprehensively summarise and visualise this variable with the SAS System.

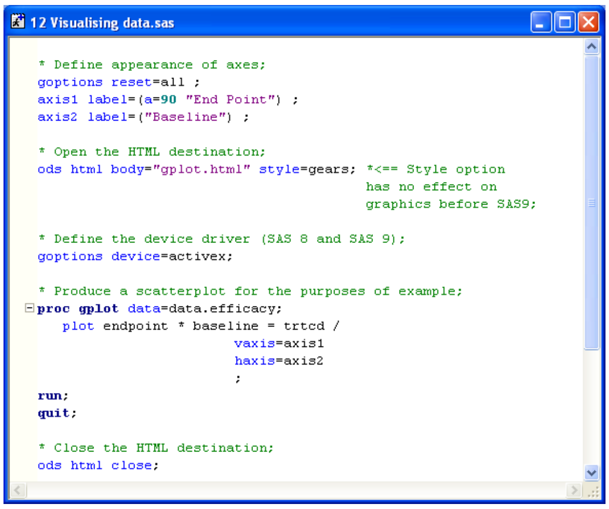


```
proc univariate data=data.efficacy;
   var endpoint;
   histogram endpoint  / normal(mu=est sigma=est);
run;
```

Figure 25: Histogram Statement in Proc Univariate

Furthermore SAS 9 builds on the interactivity of graphics introduced in SAS 8, by combining features of the output delivery system with graphics, something SAS are extending in future versions beyond SAS 9.1.3, the current version as I write.

The following code demonstrates the functionality available now. By applying a built in style from SAS 9, the graphics produced are of presentation quality with just a few lines of code.



```
* Define appearance of axes;
goptions reset=all ;
axis1 label=(a=90 "End Point") ;
axis2 label=("Baseline") ;

* Open the HTML destination;
ods html body="gplot.html" style=gears; *<== Style option
                                          has no effect on
                                          graphics before SAS9;

* Define the device driver (SAS 8 and SAS 9);
goptions device=activex;

* Produce a scatterplot for the purposes of example;
proc gplot data=data.efficacy;
   plot endpoint * baseline = trtcd /
                     vaxis=axis1
                     haxis=axis2
                     ;
run;
quit;

* Close the HTML destination;
ods html close;
```

Figure 26: Producing Interactive Graphics

The figure produced below depends only upon the appropriate device driver being installed. It is designed primarily for web usage, however also functions in other ODS formats, such as Word documents generated with ODS RTF, providing the machine used to read the document also has the device driver installed.

Note: In the code used here the ACTIVEX device driver was used. This is specific to the Windows operating systems. A JAVA device driver exists which may also be used and is suitable for cross platform usage.

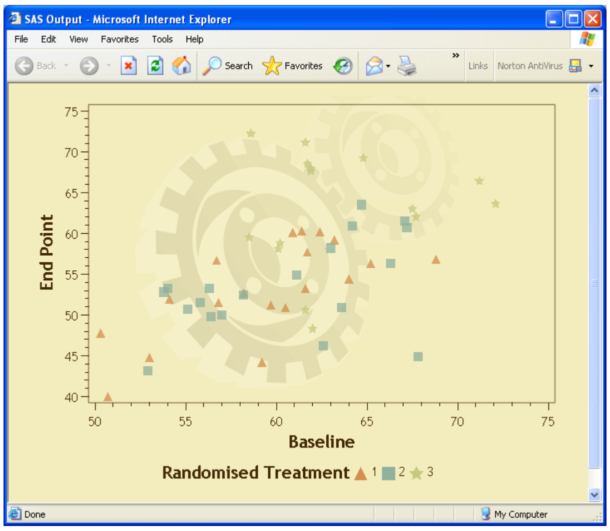

SAS Output - Microsoft Internet Explorer
End Point
Baseline
Randomised Treatment 1 2 3
Done
My Computer

Figure 27: Presentation Quality Graphical Output

Reporting

The SAS System provides three key procedures for generating reports: Proc PRINT, REPORT and TABULATE.

In my opinion the REPORT procedure is the most versatile thanks to its ability to combine the features of both PRINT and TABULATE whilst also providing programming functionality within its compute blocks.

The TABULATE procedure's key strength is its flexibility for producing tables of summary data.

ENVIRONMENTS FROM SAS INSTITUTE

In additional to the traditional environment, DMS, for programming with the SAS language, SAS Institute have developed a suit of environments that aim to harness the ability to analyse and report data quickly. Some of these environments are briefly discussed here.

SAS ENTERPRISE GUIDE

Enterprise Guide (version 3) is designed to enable users to quickly analyse and report on data on the Windows platform, moreover it allows the traditional programmer to write, submit and retrieve the log and output in a similar way to the DMS.

Furthermore Enterprise Guide has wide reaching abilities to integrate with SAS servers, allow the authoring and execution of stored processes in a project based environment. These are just a flavour of features available.

MICROSOFT OFFICE ADD-IN

The Microsoft Add-In integrates the SAS System directly within Microsoft packages such as Word and Excel. Like Enterprise Guide, SAS does not have to be installed locally and makes use of task based functionality shared with Enterprise Guide and connectivity to SAS metadata servers.

SAS SOLUTIONS

Several other SAS solutions are available which may be encountered during your experience as a Statistician or SAS programmer:

- SAS Drug Development Solution
- SAS Drug Discovery
- SAS Research Data Management
- SAS Microarray
- SAS Proteomics
- SAS Genetic Marker
- SAS/Genetics
- SAS IntelliVisor for Pharma
- SAS CRM for Pharma

CONCLUSIONS

To the intended reader, young Statistician's and Statistical Programmers, there's a lot to take in from a single read of this paper. This is also true of learning the SAS System and its capabilities itself, therefore we never stop learning.

When programming with SAS, or any other language, taking the time to plan can save time later and enables verification that what you set out to do has been achieved.

Adopt standard practices within your group or organisation. This can reduce the effort of working with each other code and helps to convey a professional attitude and approach.

From the examples provided and your existing exposure to SAS it will become clear there are many ways of doing the same thing with the SAS System. That's part of its power; it doesn't mean that one method is necessarily better than another. Consistency is the key.

Never say that you cannot achieve something with the SAS System. In my experience this is virtually never the case, sometimes you must consider the benefit gained from programming a complex task versus the effort required to achieve it.

Finally, remember that the SAS System is a tool… a *very* powerful tool, but you should never let it determine what statistics and reports are produced, but use it to determine the statistics you have asked for.

CONTACT INFORMATION

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

Author Name:	David Shannon
Company:	Amadeus Software Limited
Address:	Mulberry House, 9 Church Green, Witney, Oxon OX28 4AZ
Phone:	+44 (0) 1993 848010
Email:	david.shannon@amadeus.co.uk
Web:	www.amadeus.co.uk

Version 1.0.