SAS® MACROS - A GENTLE INTRODUCTION FOR THE FEARFUL

Ralph W. Leighton, The Hartford Insurance Group, Hartford CT

1.0 WHITHER WE GO ...

The following is an abbreviated adaptation of a Beginning Tutorials presentation made at NESUG in 1995. In its original form, the paper's emphasis was on introducing the SAS macro language, providing applications examples as motivation. In the present version, although some of the material is the same, the emphasis is shifted to posing basic application problems which are not cleanly or easily solvable without recourse to macros or macro variables.

In each of the sections, a specific application frames the discussion. Section 2.0 covers uses of global macro variables in applications. These particular ones are set using the CALL SYMPUT routine. Section 3.0 discusses simple parameter driven recyclable code using a SAS macro. In sections 4.0 and 5.0 we take the next step into complexity; we incorporate into the macros some procedural syntax to vary the form of the macro-generated SAS-program code or to short-cut laborious and repetitious coding.

The reader is not expected to be an expert in the use of macros. However, since this is not a pure tutorial, the paper does assume the reader knows what a SAS macro is and what a SAS macro variable is.

2.0 GLOBAL PARAMETERS: A BASIC USE OF MACRO VARIABLES.

Currently, my unit in the actuarial department at The Hartford operates a system which makes use of loss development "triangles". In the two-dimensional form, they can be viewed as a sequence of records (observations) having the following logical keys:

```
LINE    - Line of Business Ident Code
ITEM    - Loss Variate Ident Code
ACCYEAR - Accident Year of Loss
```

The data on each observation consists of a series of calendar valuations of the "item" referenced in the key **ITEM**:

```
LSS1, LSS2, LSS3, ...etc ...
```

These are cumulative evaluations at monthly intervals, starting with the first month of each accident year and stretching out to the current reporting month. The "item" is a quantity such as total payments to claimants, numbers of known claims, totals of settled cases to date, and total losses reported (incurred) whether paid out yet or not. This is standard property-casualty insurance data, and data like it, at annual development points, is publicly reported in such documents as the Annual Statement ("Yellow Book") and the 10-K filings for the S.E.C..

The length of these **LSS** vectors varies by month, as more and more data is added to the valuations on accident years. And depth of history is very important in lines like professional liability, which develop slowly. For the sake of discussion here, we'll assume that 1976 is the oldest year in our data repository. As of October 1996, there would be 262 months of reporting on accident year 1976 (incrementally less on other years). A vector storing all the developments for each year back to 1976 would need provisions for 262 slots or variables. By year-end 1996 the vector of valuations would be up to 264 variables. Modules in the application have DATA-steps which need arrays able to hold the vectors, and various procedures depend on knowing the current calendar or "production" date of the data.

Facilitating our application is a little SAS dataset, called **DATECARD** with a single observation containing the loss data's current production date – e.g. 1996-10 for October. The application start-up routine executes, among other things, a DATA-step that accesses **DATECARD** and uses the CALL SYMPUT routine to create global macro variables for the application:

```
DATA _NULL_;
     SET RESRVING.DATECARD;
     CALL SYMPUT('PRODMO',PRODMO);
     PROYR = FLOOR(PRODMO/100);
     CALL SYMPUT('PROYR', PROYR);
     NOYRS = PROYR - 1976 + 1;
     CALL SYMPUT('YRS', NOYRS);
     PROMON = MOD(PRODMO,100)
     CALL SYMPUT('PROMON', PROMON);
     MOS = (NOYRS - 1)*12 + PROMON;
     CALL SYMPUT('MOS', MOS);
     STOP;
RUN;
```

The variable **PRODMO** is on the **DATECARD** file and happens to be in the form of a six-digit number, like 199610, rather than a SAS Date numeric. The calendar month of the year is **PROMON** and the largest development required (that on year 1976) is represented by the calculated variable **MOS**. The CALL SYMPUT routine transfers the contents of the SAS variables to macro variables that happen (except in one case) to have the same SAS name. As long as the application is running or as long as the SAS Display Manager session is running, these macro variables are available for processing.

We can embed the values of one or another of these variables in report titles. For example, the TITLE statement:

```
TITLE2 "FORECAST DATA AS OF &prodmo";
```

will produce the title:

FORECAST DATA AS OF 199610

When a DATA-step in the application creates a permanent SAS dataset, we can also add this production date as information on the dataset's SAS label, as documenta-

tion of the file's contents:

```
DATA RESRVING.DEVLFCTR
    LABEL=
     "RS0103PC: &prodmo Devel Factors"
    KEEP= ..... etc ..... );
        :
```

Then, on a PROC DATSETS log listing, a PROC CONTENTS report or in the DIR window display, this file label will appear as:

RS0103PC: 199610 Devel Factors

(Note the required use of double quotes in the TITLE statement and the LABEL option.)

But application documentation and report cosmetics are the least of the utility of these macro variables. As noted before, processing the loss data are DATA-step programs which will analyse part of, or all of, loss triangles. Such a program might need to store all of the developments on each accident year as a one-dimensional array of data:

```
ARRAY LOSSDATA(...) LSS1-LSS???;
```

or it might need to store **all** the accident years of data for a line of business, as a two-dimensional array of data:

```
ARRAY TRIANGLE(... , ...) _TEMPORARY_;
```

Now the number of accident years and the number of development valuations will vary by production date. We could get around this obstacle by setting fixed array extents beyond immediate needs, as per:

```
ARRAY LOSSDATA(360) LSS1-LSS360;
ARRAY TRIANGLE(30,360) _TEMPORARY_;
```

This, however, is clumsy. Potentially, it opens up the application for maintenance down the road, if the extents are not generous enough. The first array, in addition, creates a plethora of data variables that are currently of no use, the unused **LSS**'s. A preferable alternative is to "variably dimension" the arrays -- a feature allowed in the language PL/1 using "controlled storage". In SAS, we accomplish the same end by making use of the macro variables created above:

```
ARRAY LOSSDATA(&mos) LSS1-LSS&mos;
ARRAY TRIANGLE(&yrs,&mos) _TEMPORARY_;
```

which, as of October 1996 data, will turn into:

```
ARRAY LOSSDATA(262) LSS1-LSS262;
ARRAY TRIANGLE(21,262) _TEMPORARY_;
```

As an alternative to indexing the accident years starting with 1 for 1976, we might rather wish to use the explicit year itself as an index. In this case we would want the array **TRIANGLE** for October 1996 to look like:

```
ARRAY TRIANGLE(1976:1996,262)
     _TEMPORARY_;
```

and this is achieved using the following coding:

```
ARRAY TRIANGLE(1976:&proyr,&mos)
     _TEMPORARY_;
```

Various DO-loops in the DATA-step code may have bounds that reference the extents of these arrays. The following code, for example, picks off the total current calendar month change of the item in the array **TRIANGLE**, using the macro variables as values:

```
ARRAY TRIANGLE(1976:&proyr,&mos)
     _TEMPORARY_;
          :
MAXDEV = &promon ;
CALCHG = 0;
DO JYR = &proyr TO 1976 BY -1;
  CALCHG = TRIANGLE(JYR,MAXDEV) -
           TRIANGLE(JYR,MAXDEV-1);
  MAXDEV = MAXDEV + 12;
  END;
```

Since the **TRIANGLE** array has been dimensioned to fit exactly the scope of data available as of the production date, the reference to macro variable **PROYR** in the DO-statement is equivalent to using the HBOUND function.

****

One word of warning as a side issue: The newcomer to SAS macros should beware of one subtlety when macro variables are used as values in DATA-steps: if you precompile your DATA-steps and RUN them later -- as opposed to the compile-load-and-go approach starting with the SAS code at usage time -- your DATA-steps will be stuck with the values of the macro variables as of **compile** time. This may not be the set of values you want....

3.0 PARAMETER DRIVEN RECYCLABLE CODE -- SIMPLE MACROS WITH PARAMETERS

The SAS macro is quite simply a program whose output is SAS Code (which is then run by the SAS Supervisor when you invoke the macro). The code it produces may be a DATA step, A PROC step, or an entire SAS program having several steps. On the other hand, a SAS macro's output can also be simply a code fragment lying within a DATA-step or a PROC step. We'll see one of these latter uses in section 5.0.

A simple common use of SAS macro's is to create parameter-driven recyclable versions of a SAS program, a use illustrated in the following example. Suppose I have a small model that maintains calendar accounting information on my insurance company in a library called **INVSTMDL**. One series of files has records with the following variables:

LINE	Line of Business Code
ITEM	Item Description
QTR0-QTR24	Item Amts by Calendar Qtr most recent 24 quarters.

LINE and ITEM are logical keys. The "item" can be: income items like earned premiums or investment income; cash-flow items like paid losses; quarter-ending balances like loss reserves; or other items like investment yields or average maturities. This data resides on eight SAS Datasets by item category, listed at the top of the next page.

Suppose someone wants to use these files for their spreadsheet application. To meet their request, I want to create importable "flat" files (Non-SAS files) from the eight SAS datasets in the table. The following code will convert the **BALSHEET** dataset:

Accounting Data Files

```
BALSHEET  - Balance Sheet
TURNOVER  - Investment Sales, Purchases
INCMSTMT  - Income Statement
CASHFLOW  - Cash-flow Statement
DURATION  - Average Pay-in and Pay-out lags
PERCENTS  - Investment Yields
INVSTDET  - Investment Portfolio Details
UWRESULT  - U/W Loss and Expense Ratio
```

```
DATA _NULL_;
  SET INVSTMDL.BALSHEET;
  FILE TOLOTUS(BALSHEET) LRECL=450;
  PUT @2 LINE  5.1
      +1 '"' ITEM  $ '"'
      +1 (QTR0-QTR24)( 13.1 );
  RETURN;
RUN;
```

(The quoting of **ITEM** assures that the recipient will be able to use the "/ File-Import-Numbers" command sequence in Lotus 1-2-3® to import the data into his worksheet. **TOLOTUS** is a FILENAME reference to a P.D.S. or Directory for the output non-SAS files.)

To write out the other seven files requires replicating the above SAS code and changing three things:

- The SAS dataset name
- The output file name (same as that of the dataset)
- The output format for variables **QTR0-QTR24**

The format changes because it happens that files **UWRESULT** and **PERCENTS** have data that are percentages to a precision of three decimal places and therefore the recipient wants them written as decimals to five decimal digits precision. **DURATION** contains average paid dates in years and the recipient needs these numbers to three decimal places precision.

Rather than clone the DATA-step code seven times to produce the other seven files, let's make a macro out of it and simply execute the macro eight times.

```
%macro MAKEFILE(file=,format=);
DATA _NULL_;
  SET INVSTMDL.&file  ;
  FILE TOLOTUS(&file) LRECL=450;
  PUT @2 LINE  5.1
      +1 '"' ITEM $ '"'
      +1 (QTR0-QTR24)( &format );
  RETURN;
RUN;
%mend MAKEFILE;
```

This little SAS macro has two key word parameters:

- **FILE** - the SAS dataset name; and
- **FORMAT** - the output format for **QTR0-QTR24**

"Key word" means we will reference the parameters' names when we execute the macro, and execute the macro, as we do below to create the eight flat files:

```
%MAKEFILE(file=BALSHEET,format=13.1)
%MAKEFILE(file=INCMSTMT,format=13.1)
%MAKEFILE(file=CASHFLOW,format=13.1)
%MAKEFILE(file=PERCENTS,format=13.5)
%MAKEFILE(file=TURNOVER,format=13.1)
%MAKEFILE(file=DURATION,format=13.3)
%MAKEFILE(file=INVSTDET,format=13.1)
%MAKEFILE(file=UWRESULT,format=13.5)
```

It does not make any difference in which order you enter key-word parameters when you execute the macro **MAKEFILE**:

```
%MAKEFILE(format=13.5,file=UWRESULT)
```

will work just as well as

```
%MAKEFILE(file=UWRESULT,format=13.5)
```

Often it is desirable to give **default values** to macro key-word parameters. In the present example. the majority of the eight files require the fixed numeric 13.1 format, and the macro **MAKEFILE** could have been set up with this default value for parameter **FORMAT**:

```
%macro MAKEFILE(file=,format=13.1);
DATA _NULL_;
     ... etc., as before ...
```

Then submitting

```
%MAKEFILE(file=BALSHEET)
```

is equivalent to:

```
%MAKEFILE(file=BALSHEET,format=13.1)
```

The ability to code default values is useful in situations where a parameter normally assumes a certain value and only assumes other values on an exception basis. For example, suppose one has a statistical routine set up to do regressions. The routine might normally use 36 months of data for most pruposes, but could use more if the operator of the routine wants it.

As an alternative, one could have coded the parameters as **Positional**, as per:

```
%macro MAKEFILE(file,format);
 DATA _NULL_;
     ... etc., as before ...
```

(Note the absence of equal signs.) An execution of this latter version of **MAKEFILE** would be:

```
%MAKEFILE(BALSHEET,13.1);
```

If you use positional parameters, the order of coding of the values **is** important. Although use of positional parameters saves some writing, the resulting SAS code is more cryptic and you cannot give them default values. Personally, I favour the key word parameters: they are forgiving of order on invocation; they do allow default values; and they are more self-documenting.

4.0 RUBBER PROGRAMS -- MAKING USE OF THE MACRO LANGUAGE IN A MACRO

I've used the jargon "rubber program" to describe the situation where, in the invocation of a macro, the particular values of the parameters affect the **syntax** of the macro-generated SAS code -- not just values of attributes or parameters in the generated SAS statements. To this

end, consider the following simple PROC PRINT:

```
TITLE "COMPETITOR INVESTMENT RESULTS";
PROC PRINT DATA=INVSTMDL.INCOME LABEL;
    BY  COMPANY  CALYEAR;
    PAGEBY CALYEAR;
    ID  LINE     LINENAME;
    VAR EARNPREM LOSSINCR UWEXPENS
        POLDIVID UWINCOME;
RUN;
```

It turns out that file **INVSTMDL.INCOME** is rather large: there are some 1,300 insurance companies (variable **COMPANY**) represented on the file with data from 1976 (variable **CALYEAR**) through the present date. Does one want to receive all that paper? Probably not as a matter of course.

We could set this program up in the manner of the macro in section 3.0, making use of a WHERE statement to ask for a single Company and Calendar Year at a time. This, however, is not really that helpful, since the end-users of this data want to look at more than one company at a time and probably at a range of calendar years. Thus a more useful subsetting WHERE would be of the form:

```
WHERE COMPANY IN ( ... list ... )
  AND CALYEAR GE first-year
  AND CALYEAR LE last-year ;
```

and this suggests the following macro **MAKEREPT** having three parameters:

```
%macro MAKEREPT(complist=,fyr=,lyr=);
TITLE "COMPETITOR INVESTMENT RESULTS";
PROC PRINT DATA=INVSTMDL.INCOME LABEL;
    WHERE COMPANY IN ( &complist )
      AND CALYEAR GE &fyr
      AND CALYEAR LE &lyr ;
    BY  COMPANY  CALYEAR;
    PAGEBY CALYEAR;
    ID  LINE     LINENAME;
    VAR EARNPREM LOSSINCR UWEXPENS
        POLDIVID UWINCOME;
RUN;
%mend MAKEREPT;
```

If I want a report for companies 32, 65, 71 and 132 for years 1983 to 1992 , I now submit the invocation:

```
%MAKEREPT(COMPLIST=32 65 71 132,
          FYR=1983,
          LYR=1992);
```

(Note the **absence** of commas in the list of companies: a comma would signal the end of the string to be assigned to **COMPLIST**, since commas separate the parameters.)

The submission above generates the SAS code below:

```
TITLE "COMPETITOR INVESTMENT RESULTS";
PROC PRINT DATA=INVSTMDL.INCOME LABEL;
    WHERE COMPANY IN ( 32 65 71 132 )
      AND CALYEAR GE 1983
      AND CALYEAR LE 1992;
          :
        ..etc...
```

and the desired report.

But suppose now I get a little fussy:

- I want ALL companies, if I omit the company list.
- If I don't enter a value for **FYR**, I want to start with the first available year automatically without hard-coding this year as a default value for **FYR**.
- Similarly, if I don't code a value **LYR**, I want output up through the last year, without coding this last available year as a default value for **LYR**.
- Finally, I want an error message printed to the SAS-Log, if **FYR** is greater than **LYR**.

This innocent complication leads to seven possible forms for the WHERE statement and one further situation in which there is no sub-setting WHERE statement at all:

```
/* All Three parameters coded */
 WHERE COMPANY IN ( ... list ... )
   AND CALYEAR GE first-year
   AND CALYEAR LE last-year ;
/* Company List omitted */
 WHERE CALYEAR GE first-year
   AND CALYEAR LE last-year ;
/* First Year Omitted */
 WHERE COMPANY IN ( ... list ... )
   AND CALYEAR LE last-year ;
/* Last Year Omitted */
 WHERE COMPANY IN ( ... list ... )
   AND CALYEAR GE First-year ;
/* Only company List coded */
 WHERE COMPANY IN ( ... list ... ) ;
/* Only First Year Coded */
 WHERE CALYEAR GE first-year ;
/* Only Last Year Coded */
 WHERE CALYEAR LE last-year ;
/* No sub-setting */
 /* ... no WHERE statement */ ;
```

All of these alternatives can be accommodated in a single macro (shown at the top of the next page). This new version, **PRNTREPT**, makes use of the following elements in the macro language's procedural syntax:

- the %LET assignment statement
- the %DO-group; and
- the %IF... %THEN... %ELSE... statement.

The **PRNTREPT** macro will conditionally produce the desired error message and will generated a PROC PRINT with all eight alternatives of the sub-setting WHERE statement. Let's walk through how the macro solves the sub-setting WHERE problem.

Following the %MACRO statement, there is a %PUT statement which writes the values of the three variables in the SAS log. The next statement,

```
%if &fyr GT &lyr
%then %put "** ERROR ** FYR > LYR";
```

(which is similar in syntax to the IF...THEN... in the DATA-step language) tests whether **FYR** is greater than **LYR**. If it is, the message

**** ERROR ** FYR > LYR**

An Example of a "Rubber Program"

```
%macro PRNTREPT(complist=,fyr=,lyr=);

%put &complist= &fyr= &lyr= ;
%if &fyr GT &lyr
%then %put "** ERROR ** FYR > LYR";

TITLE "COMPETITOR INVESTMENT RESULTS";
PROC PRINT DATA=INVSTMDL.INCOME LABEL;

%let testchar = &complist&fyr&lyr;
%if &testchar NE %then %do;
     WHERE
  %let andchar=;

  %if &complist NE %then %do;
      COMPANY IN ( &complist )
       %let andchar=AND;
  %end;

  %if &fyr NE %then %do;
    &andchar CALYEAR GT &fyr
       %let andchar=AND;
  %end;

  %if &lyr NE %then %do;
    &andchar CALYEAR LE &lyr
  %end;

       ;

%end;

   BY  COMPANY  CALYEAR;
   PAGEBY CALYEAR;
   ID  LINE     LINENAME;
   VAR EARNPREM LOSSINCR UWEXPENS
       POLDIVID UWINCOME;
 RUN;

%mend PRNTREPT;
```

will appear in the SAS log.

We now set up a macro variable **TESTCHAR** to expedite testing whether we have any parameters entered at all. The assignment statement

```
%let testchar = &complist&fyr&lyr;
```

creates **TESTCHAR** as the concatenation of the strings in the three variables **COMPLIST**, **FYR** and **LYR**. **TESTCHAR** has to have something in it for a subsetting WHERE to be coded. And that is exactly what the conditionally executed %DO-group ensures:

```
%if &testchar NE  %then %do;
     WHERE
```

This says: If **TESTCHAR** is not equal to the **null string** then we will need to begin a WHERE statement in the SAS Code.

If so, we next initialise a variable **ANDCHAR**

```
%let andchar=;
```

which will contain either nothing or the operator AND. We'll explain this one shortly. Now we test whether or not to subset on the basis of a company list. The code:

```
%if &complist NE  %then %do;
      COMPANY IN ( &complist )
    %let andchar=AND;
%end;
```

says: if the company list is not null then add the following to the WHERE statement:

```
COMPANY IN ( {value-of-complist} )
```

and place value AND in **ANDCHAR**.

The next block of code determines whether the WHERE statement needs a test for a lower bound to **CALYEAR**:

```
%if &fyr NE %then %do;
  &andchar CALYEAR GT &fyr
   %let andchar=AND;
%end;
```

The test for **CALYEAR** being greater than **FYR** is added if **FYR** is coded. But should this added code fragment be

```
AND CALYEAR GT {value-of-&fyr} ?
```

Or should it be

```
CALYEAR GT {value-of-&fyr} ?
```

Macro variable **ANDCHAR** takes care of that dilemma. If there was a sub-setting on the basis of company, we need the AND: in this instance **ANDCHAR** will have been reset to AND as a value. Otherwise we don't, and this latter case **ANDCHAR** would retain its initialisation to the null string.

The %LET statement makes sure that if **ANDCHAR** did not previously contain AND, then it does now

The next block of code is a third conditionally executed %DO-group, who purpose is to add a check in the WHERE statement for **CALYEAR** less than or equal to **LYR**. Again note the role **ANDCHAR** plays:

```
%if &lyr NE %then %do;
  &andchar CALYEAR LE &lyr
%end;
```

Now we terminate the WHERE statement with the isolated semi-colon ";" and end (with %END) the outer %DO-group used to construct the WHERE statement. The remainder of the SAS code in the macro is the invariant part of the PROC PRINT.

To execute the report for companies 23, 47 and 201 and for ALL years after 1989, we would submit:

```
%PRNTREPT(complist=23 47 201,
          fyr=1989)
```

which would produce SAS Code containing the following subsetting WHERE statement:

```
WHERE COMPANY IN ( 23 47 201 )
  AND CALYEAR GT 1989;
```

5.0 WORK-SAVING CODE FRAGMENTS AND PROGRAMS USING %DO LOOPS

The iterative %DO-loop is particularly useful in certain situations involving indexed lists of SAS variables, like the **QTR0-QTR24** on the **INCOME** file in section 3.0. Used in a

macro it provides an ability to deal with situations ordinary array overlays and DO-loop processing can't reach.

If one sets up an array overlay, as per:

```
ARRAY  QTRDATA(*) QTR0-QTR24;
```

one can use normal DATA-step language DO-loops to expedite calculations -- see, for example, the author's paper "*Working with Arrays*" (published in the 1992 NESUG Proceedings, as well as in the SUGI 1994 and 1995 Proceedings).

However, arrays won't help to reduce the code in non-executable statements like LABEL or options such as RENAME. How do you get around laborious coding like this?

```
SET INVSTMDL.BALSHEET
  (RENAME=(QTR0=BSITM0
           QTR1=BSITM1
             ... etc. ...
           QTR24=BSITM24))
       END=ENDFILE;
```

The following macro (similar to an example in the *SAS Guide to Macro Processing*) will create the RENAME sequence:

```
%macro RENAME(old=,new=,first=,last=);
    %do jnum = &first %to &last;
       &old&jnum = &new&jnum
    %end;
%mend RENAME;
```

Assuming we have compiled **RENAME**, the SET statement becomes:

```
SET INVSTMDL.INCMSTMT
   (RENAME=(
  %RENAME(old=QTR,new=BSITM,
          first=0,last=24)
    )) END=ENDFILE;
```

and this will generate the RENAME list. Note that there is **no** semicolon after the macro invocation.

Another area I've found this strategy useful is in situations where I need to process an indexed list of SAS datasets or files through the same logic -- e.g.:

```
XB2020Y0   XB2020Y1   .....   XB2020Y9
```

In the first of two examples relating to these files, a SET statement in a DATA step needs to access the ten files. This could be addressed with a macro similar to **RENAME**, with the macro then invoked within the DATA step. Or the entire DATA-step can be embedded in the macro as per:

```
%macro MERGEM;

DATA RESRVING.MERGFILE;
          ..... etc. ....
    SET %do fileid = 0 to 9;
          RESRVING.XB2020Y&fileid
        %end;    ;  /* Note 2nd semicolon */
          ..... etc. ....
RUN;

%mend MERGEM;
```

The invocation %MERGEM creates and executes the DATA step, pulling in all ten files.

The second example illustrates a situation when a %DO-loop creates, not a piece of a program, but multiple executions of the program. Suppose a DATA-step code must process each of the **XB2020Y*** files above and make an output file **ACCTRIY*** using otherwise the same logic. This is like the section 3.0 example: we set up the program in question as a simple macro with a single parameter standing for the index of the **XB2020Y*** file accessed:

```
%macro MAKEONE(fileid=);

   DATA RESRVING.ACCTRIY&fileid;
            ..... etc. ...
     SET RESRVING.XB2020Y&fileid;
                 :
            ..... etc. ...
   RUN;

%mend MAKEONE;
```

Compile **MAKEONE**. Then we can use a second macro **MAKEALL** to execute the first macro **MAKEONE** ten times, resulting in the ten executions of the DATA-step.

```
%macro MAKEALL;

   %do filenum = 0 to 9;
      %MAKEONE(fileid=&filenum)
   %end;

%mend MAKEALL;
```

Using two macros allows separate testing of the program without running all ten production legs.

6.0 MONITORING THE RESOLUTION OF MACRO VARIABLES AND MACRO CODE - "MY WHAT A TANGLED WEB WE WEAVE..."

The following final note is addressed to newcomers to macro usage: It is often helpful to see what SAS is actually executing when macros -- particularly ones like that in section 4.0 -- are invoked. To this end, there are two options SAS provides, which should probably be enabled during the development phase of applications:

- **SYMBOLGEN** - documents on the SAS-log each replacement of macro variables with values in SAS code.
- **MPRINT** - this prints out on the SAS-log each generated SAS-statement on a separate line. Unfortunately for readability, any cosmetic indentation of the code is not preserved and /* .. */ comments are dropped.

NOTES AND REFERENCES

SAS Guide to Macro Processing , ver 6, 1st Ed, published by the SAS Institute, Cary, NC.

"**Working with Arrays**" by Ralph Leighton, NESUG 1992 Proceedings; SUGI 1994 and 1995 Proceeedings.

Ralph Leighton may be reached by E-Mail at the following address: RL06950@THEHARTFORD.COM