

REUSABLE CODE: THE LAZY PROGRAMMER'S BEST FRIEND

Gary Anderson,

Hoechst Marion Roussel INC.,

Kansas City , Missouri

ABSTRACT

Clever programmers long ago discovered the advantages of borrowing code and modifying it for similar but new software needs. The 'copy and modify' approach saves time in development. However it often leads to libraries of programs that are a nightmare to maintain. The truly lazy programmer looks for ways to write his software only once, or better still, 'borrows and returns software', leaving a minimum of programs to be maintained.

This paper will be a tutorial on developing and implementing reusable SAS® code. It will describe 3 proven approaches to creating effective, reusable SAS code: Pseudo-functions, Included Code , and Exits. Each approach will be explained using a real-life project and actual SAS code. Important environmental issues will be pointed out as well. Strengths and weaknesses of each approach will be discussed.

INTRODUCTION

In the spring of '93 a team of ambitious, yet lazy programmers, was faced with a nasty engineering task. The job was to develop SAS software that would convert a diverse, nonstandard mishmash of pharmaceutical research data files into sets of 20 standard data files. The scale was large. The team would need to process 70 sets of data, known as protocols, of from 30 to 50 files each into 70 sets of standard files. Thus approximately 3,000 disparate files would need to be mapped into 1,400 standard files with each file requiring its own program. To complicate matters the source data resided on six different systems in five countries. Furthermore, within each of the six systems the file formats and variable definitions often varied dramatically from protocol to protocol.

At the completion of the project each of the standard files would contain an important category of research data for each protocol. For each protocol there would then exist a file of lab results (LAB), a file of physical exam results (PHYSEXAM) and 18 more standard files.

The purpose for all this standardization was to allow for quick and easy analysis of data across protocols. Thus, when the project was finished an analyst could concatenate all 70 ADVERSE files, or any meaningful subset of the 70, and be assured that all variables would have the same meaning, format and name. He could then analyze the combined data as one. Where a cross protocol analysis could take weeks using the source data(if it could be reasonably done at all), the same analysis could be reduced to minutes on the standardized data.

Clearly, if the team were to keep it's sanity, and it's social schedule some new ideas would be in order.

A common approach to creating such a large number of programs is to 'copy-and-modify'. This approach had a great deal of appeal in this project for though there was a lot of disparity in the source data, there would still be a lot of processing that would need to be done in a standard way. Once a set of programs was in production for the first protocol, it would be expedient to copy those programs and modify them for the second protocol and so on. However there were 3 distinct problems with this approach.

1. Consistency. With copy-and-modify there could be no way to be sure that a program segment that performed a standard task across all protocols would in fact remain unchanged after being replicated 70 times.

2. Maintenance. If an algorithm changed or an

error were found in a program segment, it would be necessary to track it down in as many as 70 programs, fix it, up to 70 times, and verify correctness up to 70 times. A potential nightmare.

3. Clutter. As the project grew, large amounts of common program segments would be spawned with each copy-and-modify. Programs would be cluttered with code that performed a standard task but filled up programs with lines and lines of common code. This would leave the programmer to wade through each program section and determine what it did, always with the nagging question, 'Is this chunk of code really, exactly like the one from the last protocol'?

THE SOLUTION: REUSABLE CODE

To tackle the three problems of copy-and-modify our heroes chose a strategy of, wherever possible, writing code once and then calling it from a common library at run time - i.e. REUSEABLE CODE! Thus, rather than having up to 70 copies of standard program segments, there would be only one.

This solved all three problems in one fell swoop. By removing code to a single location it could be known with certainty that all programs that used that code would process data in exactly the same way. If the code needed maintenance, the change would be made in one place, tested in one place and then be ready for use everywhere. Program clutter would be reduced by changing program segments to a one line call to the production library. Essentially the code segments would become 'black boxes'.

Reusable code took three forms:

- 1) Pseudo-functions. These were really SAS macros that behaved like SAS functions.
- 2) Included code. These were program fragments that performed the same duties in all programs regardless of which protocol was involved.
- 3) Included code with exits. This was a more advanced, more flexible way to use included

code.

How did this actually happen in SAS? The next three sections will describe the 'nuts-and-bolts' of psuedo-functions and included code.

PSEUDO-FUNCTIONS:

A pseudo-function is a SAS macro stored in a production library. Its power comes from two sources. First, it can be invoked with one line of code from a SAS program. Second, whatever algorithm it is using will be the same for every program that uses it. It is called a pseudo-function because it behaves just like a SAS function. It takes one or more input parameters, performs a single, well defined task and then returns a value.

To invoke the pseudo-function NUMBER, listed in Appendix A, a programmer simply codes as in exhibit 1:

```
Some SAS code

%NUMBER(CLEANUM, DIRTYNUM);

Some more SAS code;
```

1 Psuedo-function invocation

When SAS gets to NUMBER it will go to the proper library, pull out the macro NUMBER.SAS, process the value stored in the variable DIRTYNUM and return a pure numeric value in CLEANUM. In this example the first parameter is the result. This is not required, however to create the feel of a function, a standard was set that the first parameter would be the result and input parameters would be in positions 2 through n.

NUMBER was created to solve a common problem in research data - that is the recording of strange symbols (>,<,...) and verbiage (LT, GT, ONE...) in what is really a numeric result. In this

project, rules were set up defining exactly how the 'character debris' should be removed and a pure, numeric field returned. It was important that every programmer 'clean' such data fields in exactly the same way. It was also important not to waste time having each programmer develop his own algorithm to implement these rules. As additional rules were needed they could be added in one place and be immediately available to all programs. Thus, NUMBER was created and a fairly elaborate algorithm became a simple one-line function.

Similar psuedo-functions were created to produce standard dates, calculate age the same way, calculate time between events uniformly etc. In all, over 40 functions were created - some with only 1 line of SAS code, some extending over 4 pages.

But the picture is not complete! How does SAS know where to find NUMBER? Some minor setup is required. First, as with any MACRO library, all of the pseudo-functions are put in one directory as a collection of files with each file given the same name as the MACRO inside. Part of the actual project directory is shown in Exhibit 2a.

Second, a CONFIG file is created with a SASAUTOS option defined. This tells SAS where to look for MACROs. Exhibit 2b. shows a sample CONFIG.SAS file. The libname SASMACRO was the actual libname assigned to the directory shown in Exhibit 2b.

Third a file that invokes SAS is created that includes a parameter telling SAS to process the CONFIG file. The actual invocation of SAS, used in the project is shown in Exhibit 2c.

In the project environment these linkages were programmed into DCL command files so that the linkages were transparent and automatic.

a. The SAS macro directory.

```
ACTTM.SAS    AGE.SAS
CDSDATE.SAS  COUNT.SAS
NUMBER.SAS   SETUP.SAS
.....
```

b. The CONFIG.SAS file

```
/NONEWS
/MAUTOSOURCE
/NOSYMBOLGEN
/SASAUTOS=(SASMACRO,GIDBMAC,
SASAUTOS)
/INITSTMT=%STARTUP
.....
```

c. The invocation of SAS with CONFIG.SAS

```
SAS/CONFIG=
  DRUGUTIL.CONFIG.SAS
.....
```

NOTE: SASMACRO is actually a VAX logical assigned using a DCL command:

```
$ DEFINE SASMACRO
  [NM08597.MDL73147. SASMACRO]
```

SAS on the VAX allows one to substitute logicals for libnames. In other environments, a typical LIBNAME pointing to the MACRO library will need to be defined.

DRUGUTIL is a VAX logical pointing to a directory where CONFIG.SAS resides.

2. Environment setup

INCLUDED CODE

Included code is really a collection of program fragments that are used over and over again. It could include a partial DATA step or a full blown program with numerous PROC and DATA steps. These program fragments are coded up and stored in a common program directory. A filename is associated with the directory and is used to bring in the needed piece of code. Because the team had some experience with COBOL the directory was called COPYLIB.

The basic approach to invoking included code is illustrated in Exhibit 3.

```
/* PROGRAM: NONSTUDY */  
  
Some SAS code  
.  
.  
%INCLUDE COPYLIB(NS_CDMS)
```

3 NONSTUDY program

Exhibit 3 can be thought of as a full blown SAS program - processing NONSTUDY medication data. The first part of the program is specific to the protocol. This will be because the source data has peculiar variables, formats or file structure. Once the protocol specific data has been processed to a point where it is in a standard form, final processing can be completed by a call to the COPYLIB directory to go get a copy of NS_CDMS.SAS, slap it on the end of the NONSTUDY program and complete the processing.

The important point here is not to look at the specifics of these programs - for each is really just a chunk of SAS code - but rather to realize that the code in COPYLIB exists in one and only one place and is used to finish each and every one of the 70 NONSTUDY programs.

As with psuedo-functions, the included code member gives the assurance that the code therein will always be processed the same way and that when maintenance is needed, it can be done quickly and easily.

Appendix 3 shows a live example of included code. This code was called by a COPYLIB member. This illustrates that included code can be nested (The number of nestings is system dependent). This particular piece of code only has 4 SAS commands but it was used in the last DATA step to ensure that only the desired variables were kept and that they all had standard lengths, formats and labels.

The team found included code to be most helpful. However, occasionally they found that that they needed to make a slight change in the included code member but wanted to preserve the advantages of the standardized code. This led to the use of a clever idea that came to be known as exits.

INCLUDED CODE: EXITS

The big idea with exits is to identify points in a reuseable program where modifications might be required. Then at run time, if needed, actual code is passed to the program and inserted at each of the points in the program. The idea is represented in Exhibit 4.

```
/* PROGRAM: EXHIBIT4 */  
  
Some SAS Code.  
.  
.  
&EXIT1  
Some more SAS Code.  
.  
.  
&EXIT2  
Even more SAS Code.  
.  
.
```

4 Included code with exits.

Note that in Exhibit 4 the exits look like SAS macro variables. In fact that is what they are. Also note that, though only two exits are shown, the number is limitless.

The exits in Exhibit 4 are defined in the calling program using %STR with a %LET. Consider Exhibit 5.

```

/* PROGRAM: EXHIBIT5 */

%LET EXIT1 = %STR(

    PROC SORT DATA=DRUG.DOSE
              OUT =WORK.DOSE;
    BY PATIENT;
    RUN;

);

%LET EXIT2 = %STR();

%INCLUDE COPYLIB(EXHIBIT4);

```

5 Included code, calling program

The program in Exhibit 5 defines &EXIT1 as a string. In this case the string is a simple PROC SORT. It could however be numerous DATA and PROC steps. When the program EXHIBIT4 is called by EXHIBIT5, it will insert the string of SAS code at the point of &EXIT1 and process the PROC SORT as part of program EXHIBIT4.

Note the &EXIT2 has been defined as null in FIGURE5, indicating that no special processing is required. It is important to note that once a program has an exit defined within it, it must be given at least a null value as in &EXIT2 above.

Examples of actual code using exits, are given in Appendices 4 and 5. Notice that the calling program has only two entities: A set of exit definitions and the invocation of the standard program.

CONCLUSION

The advantages of reuseable code have been stressed above. However it is no silver bullet. Several considerations are in order before

implementing reuseable code.

Be aware that it will take a lot of time up front - perhaps two to three times as much time to develop and test a piece of reuseable code, than to create a basic program. Time will be spent figuring out just which parts are reuseable and which parts are not. Consequently, one should anticipate a significant amount of common programming before implementing reusable code. In our project, if there had been only five or six protocols, it would have been better to simply copy, modify and move on. However somewhere before we reached a dozen protocols the pay off became large.

Also be aware that reading program code could be confusing for those that come after you. Following program logic and coming to a %INCLUDE command can feel like turning a corner and bumping into a brick wall. The storage location of the included code should be easily accessible to the following programmers so that he or she can easily read it. Some explanation of the include code, in the calling program, would also be very helpful.

At times the question arose, should this piece of code be a function or should it be treated as common/reusable program code? The team's guidelines were these: If there was only one task involved where a single value was calculated, make it a psuedo-function, otherwise make it common code. We also found that by invoking the option SOURCE2, the SAS log was much more informative when using included code than with a psuedo-function.

ACKNOWLEDGEMENTS

SAS is a registered trademark of SAS Institute Inc. in the USA and other countries. ® indicates USA registration.

The author would like to thank Glenn Becker, a member of the project team, for his thoughtfulness and inspiration.

APPENDIX 1.

```

macro number(outvar, invar);
  * Take character variable invar, which supposedly contains a number
  and convert it to a number. The %NUMBER macro correctly interprets
  1. Any legal SAS numbers.
  2. Numbers with extraneous characters: >6 5+ 1,000,000
  3. Numbers preceded or followed by extraneous words: "5 PILLS"
     "APPROXIMATELY 23"
  4. Fractions and Mixed Fractions: 3/4 6 1/2
  5. Words: ZERO to TEN, upper or lower case:

  ** Number was inspired by the contents of CDS ADV EVNT variables
     DUR_EVNT and DCSKEVNT, which are supposed to be numeric:

  * Not indexing multiple nested IF-THEN-DO because there are
     so many of them:

  * Try to read as ordinary number:
  %OUTVAR = INPUT(%INVAR, ?? 12.);

  IF %OUTVAR EQ . AND
  /* no obvious reason why */ %INVAR NOT IN ('.', '. ') THEN DO;

    * Try reading without extraneous characters:
    * Using TRANSLATE instead of COMPRESS so that 1/3 is not
      misinterpreted as 13:
    %OUTVAR = INPUT(TRANSLATE(%INVAR, ' ', '##c##/.,;()'''' ), ?? 12.);

  IF %OUTVAR /* still */ EQ . THEN DO;
    * Try reading as a fraction or number followed by words:

    * Try reading first word as a number:
    %OUTVAR = INPUT( SCAN(%INVAR, 1, ' '), ?? 12.);

    * If first word was a valid number, a fraction may follow,
      else first word might be a fraction:

    LENGTH _FRACT $12;
    _FRACT = SCAN(%INVAR, 1+(%OUTVAR NE .), ' ');
    _NUMERAT = INPUT( SCAN(_FRACT, 1, '/'), ?? 12.);
    _DENOM = INPUT( SCAN(_FRACT, 2, '/'), ?? 12.);
    IF N(_NUMERAT, _DENOM) EQ 2 THEN
      /* SUM means missing %OUTVAR OR */
      %OUTVAR = SUM(%OUTVAR, _NUMERAT / _DENOM);

  IF %OUTVAR /* still */ EQ . THEN DO;
    * Try interpreting a range of two numbers, e.g. 5-10.
      Do not assume high comes last. ;

    _RANGE1 = INPUT( SCAN(%INVAR, 1, '-'), ?? 12.);
    _RANGE2 = INPUT( SCAN(%INVAR, 2, '-'), ?? 12.);
    IF N(_RANGE1, _RANGE2) EQ 2 THEN
      %OUTVAR = MAX(_RANGE1, _RANGE2);

  IF %OUTVAR /* still */ EQ . THEN DO;
    * Try reading each word as a number:
    LENGTH _WORD $12;
    N = 1;
    DO UNTIL(%OUTVAR NE . OR _WORD EQ '');
      _WORD = SCAN(%INVAR, N);
      %OUTVAR = INPUT(_WORD, ?? 12.);
      N = 1;
    END;

  IF %OUTVAR /* still */ EQ . THEN DO;
    * Try reading as words ZERO-TEN:
    * _FRACT is still the first word of %INVAR:
    _FRACT = UPCASE(_FRACT);

    DO N = 0 TO 10;
      IF _FRACT = UPCASE(PUT(N, WORDS.)) THEN %OUTVAR = N;
    END;

  IF %OUTVAR /* still */ EQ . THEN DO;
    * Try reading as hours:minutes or minutes:seconds:

    _HOUR = INPUT( SCAN(%INVAR, 1, ':'), ?? 12.);
    _MINSC = INPUT( SCAN(%INVAR, 2, ':'), ?? 12.);

    IF N(_HOUR, _MINSC) EQ 2 THEN
      %OUTVAR = SUM(_HOUR, _MINSC/60);

  END;
  * Of Hours:minutes, minutes:seconds;
  END;
  * Of ZERO-TEN;
  END;
  * Of Fraction or extra words;
  END;
  * Of Reading each word;
  END;
  * Of looking for a range;
  END;
  * Of Not regular number;
  * DROP _N _FRACT _NUMERAT _DENOM _WORD _RANGE1 _RANGE2;
  %mend;

```

APPENDIX 2.

```

**** -----
****      NONSTUDY
****      ATTRIBUTES FOR GIDS NON-STUDY MEDICATION FILE
****
****      CREATED BY: GLEN BECHER      CM      4/18/93
****      MODIFIED BY:                  CM
**** -----

```

```

KEEP PROTOCOL PATIENT RX NUM PER_NUM
NS_ACTIM NS_DRUG NS_GENER
NS_ATC1 - NS_ATC5
NS_STRTD NS_STRTH NS_STPDT NS_STPTM
NS_CONT NS_DOSE NS_INDIC NS_MEDTH
;

```

```

LENGTH PROTOCOL PATIENT      $20
RX NUM PER_NUM                8
NS_ACTIM                      8
NS_DRUG NS_GENER              $40
NS_ATC1 - NS_ATC5             $50
NS_STRTD                      $8
NS_STRTH                      $5
NS_STPDT                      $8
NS_STPTM                      $5
NS_CONT                        8
NS_DOSE                       $30
NS_INDIC                      $40
NS_MEDTH                      8
;

```

```

FORMAT RX NUM PER_NUM          2.
NS_ACTIM                      8.2
NS_CONT                       NY.
NS_MEDTH                      NS_MEDTH. ;

```

```

LABEL PROTOCOL = 'Protocol Number'
PATIENT = 'Patient Number'
RX NUM = 'Patient Treatment Number'
PER_NUM = 'Treatment Period Number'
NS_ACTIM = 'Actual Medication Administration Time'
NS_DRUG = 'Medication Name (per CRF)'
NS_GENER = 'Generic Name of Med (per IMED)'
NS_ATC1 = 'ATC classification text'
NS_ATC2 = 'ATC classification text'
NS_ATC3 = 'ATC classification text'
NS_ATC4 = 'ATC classification text'
NS_ATC5 = 'ATC classification text'
NS_STRTD = 'Medication Start Date'
NS_STRTH = 'Medication Start Time'
NS_STPDT = 'Medication Stop Date'
NS_STPTM = 'Medication Stop Time'
NS_CONT = 'Medication Continues'
NS_DOSE = 'Dose Size and Route'
NS_INDIC = 'Indication Prompting Medication'
NS_MEDTH = 'Med Before, During, or After Study';

```

APPENDIX 3.

```

*****
*** AUTHOR: GARY ANDERSON DATE: JAN 09,1995 ***
*** FILENAME: POSTPROC.SAS ***
*** DIR: (LOOK UP IN KEDIT) ***
*** DESCRIPTION: ADD NEW VARIABLES TO EXISTING FILES BY POST PROCESSING ***
*****;

%GLOBAL EXIT0 EXIT1 EXIT2 GROUPNAM;

%LET EXIT0 = %STR(
    PROC SORT DATA = DRUGOUT.DOSE OUT = WORK.DOSE;
    BY PATIENT RX_NUM PER_NUM DS_ACTM;
    RUN;

    DATA WORK.DOSE;
    SET WORK.DOSE;
    BY PATIENT RX_NUM PER_NUM DS_ACTM;
    IF FIRST.PATIENT;
    RUN;

    PROC PRINT;
    RUN %MPRINT;
);

%LET EXIT1 = %STR(
    /* NO SUBSETTING NEEDED */
);

%LET EXIT2 = %STR(
    PD DOSE = DS ADOSE;
    DS_UNIT = 'MG/KG';

    IF DS_DRUG IN ('ATIVAN','BENDRYL','DECADRON',
        'DEXAMETHASONE','DIPHENHYDRAMINE',
        'LORAZEPAM','REGLAN','TORECAN',
        'THIETHYLPERAZINE')
    THEN DS_DRUG = 'SAT';

);

%LET GROUPNAM = CHEMO-IV; /* SHOULD NOT BE NEEDED */
%INCLUDE COPYLIB(ISSDOSE2); /* USING VERSION 2 NOT 1 */

```

APPENDIX 4.

```
*****
*** AUTHOR: GARY ANDERSON DATE: FEBRUARY 2,1995 ***
*** FILENAME: ISSDOSE2.SAS ***
*** DIR: DIA0:[NMS1165.SDL73147.COPYLIB] ***
*** DESCRIPTION: ADD 2 NEW VARS TO AN EXISTING PERIOD FILE. ***
*** USED WHEN ISSDOSE1 WILL NOT WORK. ***
DATA DRUGINP.PERRACK;
SET DRUGOUT.PERIOD;
RUN;

&EXIT0;

PROC SORT DATA = WORK.DOSE
(KEEP = PATIENT RX_NUM PER_NUM DS_SCHEM DS_DRUG DS_DOSE DS_ROUTE
DS_UNIT DS_ADOSE DS_WGTKG
RENAME = (DS_DOSE = PD_DOSE DS_ROUTE = PD_ROUTE))
OUT = DOSESUB(KEEP=PATIENT RX_NUM PER_NUM DS_DRUG PD_DOSE DS_UNIT
PD_ROUTE DS_ADOSE DS_WGTKG)
;

&EXIT1;

BY PATIENT RX_NUM PER_NUM;
RUN;

PROC PRINT;
RUN &MPRINT;
TITLE 'DOSE FILE W/REMOVED DOSE AND ROUTE';

DATA DOSESUB;
SET DOSESUB;
BY PATIENT RX_NUM PER_NUM;

&EXIT2;

* Next group should cover Oral Chemo and PO.;
* GROUPNAM should be defined, but works this way.;

IF (DS_UNIT EQ 'MG') AND ("&GROUPNAM" EQ '') THEN DO;
SELECT;
WHEN ( PD_DOSE LE 0.0 )
;
```

MORE SAS CODE