

Techniques for writing robust SASTM macros

Martin Gregory, Merck Serono, Darmstadt, Germany

1 Abstract

In complex applications it is important that individual components are robust and interact in a predictable manner. A robust program should be well documented and tested, should validate its inputs, handle failure in a controlled and elegant way, should appropriately notify the calling programmer or program of the outcome of execution and clean up after itself. This paper describes specific techniques for implementing these concepts when building applications using the SAS macro language. While a certain investment is involved in applying the techniques, they result in more reliable and easier to maintain systems with a lower support requirement.

2 Introduction

The goal of this paper is to describe techniques for writing SAS macros which are robust and reliable. We define robustness for programs in general to mean that the program

1. be well documented,
2. be well tested,
3. validate its inputs,
4. handle failures in a controlled way,
5. notify the caller or calling program of outcome,
6. clean up after itself.

In a sense, robust programs are self-contained and may be used by programmers who will never have any contact with the author of the program other than using the program itself.

We concentrate primarily on specifics of achieving these properties with the SAS Macro Language and therefore only touch briefly on the topics of documentation and testing.

Achieving robustness requires additional effort in designing and implementing a macro so this cost needs to be weighed against the benefits of robustness. If a macro is part of a system of macros, or is used by many other programmers then the additional effort is strongly recommended. Robustness minimizes support by ensuring that the macro is easy to use and interacts clearly with the caller.

3 The SAS Macro Language

SAS programmers are accustomed to thinking of the SAS macro language as the way to write functions in SAS. This is, however, only true with respect to the SAS macro language itself. It will be helpful to review some features of the language as this will aid in explaining the techniques described in this paper. Additionally, we introduce a categorization of macros directly related to the techniques.

According to the SAS Macro Language Reference [1], the SAS macro facility is

a tool for extending and customizing the SAS System and for *reducing the amount of text you must enter* to do common tasks.

The SAS macro language itself produces text which is passed to the SAS supervisor for compilation and execution.¹ This text may be produced from a macro variable or from a so-called macro, a user-defined function. Since there is effectively no restriction on where a macro call can be placed and, therefore, where the generated text is inserted, both the writer and caller of macros need to be aware of the context in which the macro may be used. For this reason we identify three categories of macros and introduce a corresponding naming convention.

¹We exclude discussion of the much more complex interaction between SAS macro language and SCL. See [2] for details

The first category, which we shall call *full-step macros*, generates one or more complete SAS data or procedure steps. Full-step macros can be considered the equivalent of SAS procedures. A trivial example of this type is the macro *prt* which sorts a data set and then prints it:

```
%macro prt(data,by,vars) ;
  proc sort data=&data out=prt_temp ;
    by &by ;
  run ;
  proc print data=prt_temp split ;
    var &vars ;
  run ;
%mend ;
```

The second category of macro, which we shall call *in-step macros*², generates a fragment of code for a step, i.e. it does not include a step boundary [4]. This type of macro might generate a variable name, an expression for an *if* statement or one or more complete statements. It does not, however, include a step boundary. An example is the macro *read_ydm* which reads date values in the format YYYYMMDD, recognizing when fewer than 8 characters are present in the input³:

```
%macro read_ymd(in,out) ;
  if length(&in)=8 then &out=input(&in,? yymmdd8.) ;
  else put "NOTE: possible invalid date: " _n_ = &in= ;
%mend ;
```

The third category of macro, which we shall refer to as a *pure macro*, does not generate any code for a SAS data or proc step. Rather it contains only macro statements or functions and produces a value to be used by another macro. Examples of this are the *verify(arg,target)* and *datatyp(arg)* macros delivered with SAS which, respectively, return the position of the first character in the argument that is not in the target value and determine whether the argument is numeric or character.

4 Overview of techniques for writing robust macros

The basic approach to writing robust macros is to add code which first checks whether it is safe to execute statements or steps and which, after execution, checks whether the code has executed correctly. Depending on the results of a check, the macro may be prematurely terminated. Independent of the results of checks, the macro passes information about its execution status back to the caller. In addition to these purely coding approaches, further techniques relate to the user interface and to testing. The specific techniques we describe can be broken down into the following categories:

- defining parameters;
- parameter validation;
- checking of return codes for global statements, data steps, procedure steps and macro calls, taking appropriate action as necessary;
- returning status to the caller;
- restoring the environment;
- testing; and
- documentation.

Each of these may be applied in a more or less detailed way depending on the circumstances and risks involved. While the techniques apply to all three categories of macro, we first discuss the techniques as they apply to *full-step macros* and then examine special considerations for *in-step macros*. A discussion of *pure macros* is outside the scope of this paper.

²SAS 9.2 introduces the FCMP procedure [3] which allows a programmer to write functions for use in the data step and in selected procedures which, in time, may render in-step macros obsolete.

³the informat yymmdd8. reads a 6 digit date such as 200101 without complaint as 1 January 2020

5 An illustrative example: finding duplicates

We illustrate the use of the techniques by applying them to a simple but realistic macro. Not all of the techniques are actually necessary for this particular macro, we note when this is the case.

A common task when examining data is to identify duplicates. While SAS introduced the *dupout* option for the *sort* procedure in release 9, the behaviour is not very helpful for comparing the duplicates as one of each duplicate is written to the primary output data set and the remaining to the duplicates output data set. The macro *finddups* provides an alternative, writing all duplicates to the output data set:

```
%macro finddups(data=_last_, out=_dups, by=, where=) ;
    %local i lastby ;

    proc sort data=&data out=&out ;
        by &by ;
        %if (%quote(&where) ^= %str()) %then %do ;
            where &where ;
        %end ;
    run ;

    /* determine the last by variable for use with first. and last. */
    %let lastby=%scan(&by,-1,%str( ));

    data &out ;
        set &out end=__end;
        by &by ;
        if first.&lastby + last.&lastby < 2 then output &out ;
    run ;
%mend ;
```

The complete robust version is presented in Appendix A.

6 Defining parameters

The robustness of a program can be enhanced by making sure that the program is easy to use. A well-designed user interface can reduce the likelihood of error due to incorrect use of the program. For most SAS macros, there are three different aspects to the user interface: the documentation, the parameters passed to the macro, and the messages it returns to the user. In this section we address the question of defining parameters.

The macro language allows parameters to be defined as positional or named. There are several advantages to named parameters:

1. a default value may be specified, but see section 7 below;
2. the order in which the parameters are specified by the caller is irrelevant. This contributes to ease of use;
3. especially if there are a large number of parameters, the names make it easier to assign the corresponding value to a parameter - one does not have to count commas in order to determine which parameter is being set;
4. for a programmer who has to take over maintenance of a program it is trivial to see which value corresponds to which parameter;
5. new functionality can be added to a macro without requiring existing calls to the macro to be changed. Additional parameters need only have a default which invokes the previous behaviour.

Additional clues can be provided by choosing suitable names for parameters. A good guide is to follow the naming conventions used for options and statements in SAS procedures, for example *data* for a primary input data set, *out* for a primary output data set, *infile* for a primary input external file, *file* for a primary output external file, *var* for a list of variables, or *by* for a list of variables to use for sorting.

For parameters which take a limited number of values, ease of use can be promoted by consistency in the definition of the parameters. For example, parameters which act as switches should be consistently defined as having values 1/0, Y/N or YES/NO. When such parameters are coded with character values, case should not be important.

Positional parameters also have their place, in particular when the number of parameters is very small. Precisely where the boundary lies depends on the individual case, but for more than 3 parameters it is worthwhile considering using only named parameters.

7 Parameter validation

There are three different approaches to the validation of parameters:

1. no validation. In this case we assume that if a problem arises as a result of an invalid parameter we can then handle the error;
2. formal parameter evaluation. By this we mean that the macro checks that all required parameters have been supplied and that the supplied values are those expected, for example, a valid data set name, one of a list of expected values, properly typed values;
3. format and content validation. By this we mean that in addition to the formal parameter validation, the existence or appropriateness of the values specified in the parameters are checked. For example, we may check for existence or non-existence, as appropriate, of a data set named in a parameter, that a value is numeric, or that a numeric value is in an expected range.

There are situations in which each of these approaches are valid. For each individual parameter the impact of an invalid parameter must be balanced against the effort of coding the validation. Validation prior to running any code also allows more specific error messages and return codes. Additionally, a different set of criteria may be used for the three different categories of macros.

Let us consider the first step of our *finddups* example:

```
proc sort data=&data out=&out ;
  by &by ;
  %if (%quote(&where) ^= %str()) %then %do ;
    where &where ;
  %end ;
run ;
```

If an invalid data set name is specified, SAS will return an error so, providing we trap the error as described in section 8 below, there is no real need to check that the data set name passed via the parameter is a valid SAS data set name. If the macro is complex or we wish to provide more specific information about the error or we wish to reduce the amount of text in the SAS log, however, it is worthwhile checking the parameters.

A general way to check that all required parameters have a value is to loop over a list of these parameters, checking that they have been specified. In the case of *finddups*:

```
%local _pi _params _param rc;
%let _params=.data.out.by.;
%let _pi=1;
%do %while(%scan(&_params,&_pi,.)^=%str()) ;
  %let _param=%scan(&_params,&_pi,.);
  %if %quote(&&&_param) = %str() %then %do ;
    %put ERROR: &sysmacroname: Parameter %upcase(&_param) is required.;
    %let rc=1;
  %end;
  %let _pi=%eval(&_pi+1) ;
%end;
%if &rc=1 %then %return ;
```

We use the local macro variable *rc* to flag whether any parameter is missing in order to report on all which are missing. Assuming that we also want to check the existence of the input data set in our example, we can extend the code as follows:

```
%do %while(%scan(&_params,&_pi,.)^=%str()) ;
  %let _param=%scan(&_params,&_pi,.);
  %if %quote(&&&_param) = %str() %then %do ;
    %put ERROR: &sysmacroname: Parameter %upcase(&_param) is required.;
    %let rc=1;
  %end;
  %let _pi=%eval(&_pi+1) ;
%end;
```

```

        %end;
    %else %if &_param=data %then %do ;
        %if %sysfunc(exist(&&&_param))=0 %then %do ;
            %put ERROR: &sysmacroname: %upcase(&&&_param) does not exist.;
            %let rc=2;
        %end ;
        %let _pi=%eval(&_pi+1) ;
    %end;
    %if &rc=1 or &rc=2 %then %return ;

```

We use the `%quote()` function to avoid an invalid logical expression when the parameter is actually missing. A more general approach, which would also prevent special characters in the argument from causing unexpected results, is to first quote the parameter values using `%nrquote()`:

```

%let &_param=%nrquote(&&&_param) ;
%if &&&_param = %str() %then %do ;

```

Note also that we use `%str()` in the condition to explicitly indicate that we are comparing to a missing value.

The foregoing example uses macro statements to validate the parameters. Checking that a value is specified, that the value is numeric or character, or that the value is in a specific list or range of values can be easily done in this manner. The `%sysfunc()` function is particularly useful in this context as it allows the calling of data step and SCL functions directly as illustrated by the previous example for checking the existence of a data set. With a large number of parameters, or with more complex checks, however, one should consider using a data step. Especially if there are a large number of function calls, macro code using the `%sysfunc()` function becomes difficult to read. All functions which may be used with `%sysfunc()` may be used in the data step. Handling parameters which require similar checks can be simplified by the use of data step arrays.

When using the data step, one should use the `trim()` or `strip()` functions when writing values to macro variables as, contrary to what the documentation suggests, macro variables created with `symput()` have trailing blanks which may be significant in some situations. For example:

```

24      data _null_ ;
25          value='abc      ' ;
26          call symput('mvar',value) ;
27          call symput('mvar',trim(value)) ;
28      run ;
29
30      %let mvar=&mvar.abc ;
31      %let mvar=&mvar.abc ;
32      %put mvar = &mvar ;
mvar = abc      abc
33      %put mvar = &mvar ;
mvar = abcabc

```

A further point to consider is the handling of default values. Although the author of a macro may specify default values, the caller can set them to missing. During parameter validation, the default values may be reset if the caller has explicitly set them to missing and a missing value would make no sense in the context.

8 Checking and reacting to outcomes

So far we have discussed actions which are taken before any of the code which carries out the actual task of the macro is run. We now turn to the checking and reacting to the outcome of executing this code. The primary technique is to check that each global statement, data step or procedure step has run correctly before continuing with subsequent steps.

First we consider checking whether code has run without errors. SAS provides a number of automatic macro variables which may be used to check whether a step or global statement executed successfully. These are

syserr containing a return code status set by some SAS procedures and the data step;

sysfilrc containing the return code from the last *filename* statement;

syslibrc containing the return code from the last *libname* statement.

In all three cases, a return code of 0 signifies success. For *syserr*, values less than or equal to 4 signify warnings, while values above 4 are errors. The meaning of the value returned depends on the procedure and is generally not documented. As an application of the *syserr* macro variable, consider the following extension of our sample macro:

```
proc sort data=&data out=&out ;
  by &by ;
  %if (%quote(&where) ^= %str()) %then %do ;
    where &where ;
  %end ;
run ;
%if &syserr > 4 %then %do ;
  %put ERROR: &sysmacroname: error sorting the dataset. ;
  %return ;
%end;
```

If there is an error in the sort step, the macro exits prematurely, avoiding a needless execution of the subsequent data step.

Under certain circumstances *syserr* contains a non-zero value following errors in global statements such as *axis*, *legend*, *pattern* and *symbol*, but this behaviour is not documented. SAS Note 4420 [5] documents a similar issue with the ODS statement.

In certain circumstances, the value of the *syserr* macro variables may be unreliable. SAS Note 620 [6] reports that incorrect values for &syserr may be returned from Graphics procedures. For example, the author has verified that, in both SAS 8.2 and SAS 9.1.3, the GSLIDE procedure fails when output is routed using the *gsfname* option to a file reference containing an invalid path.

```
1      filename g "/a/non/existent/file" ;
2      goptions gsfname=g ;
3      proc gslide ;
4          note height=10
5              justify=center 'bad gsfname';
6      run;
ERROR: /a/non/existent/file does not exist.
ERROR: Unable to open graphics device I/O.
ERROR: Unable to initialize graphics device.
7      quit;
8      %put syserr=&syserr ;
syserr=0
```

A further example of an incorrect return code involving keeping non-existent variables is documented in SAS Note 8522 [7].

So far in this section we have considered actions taken when errors occur. The same approach can be applied to any outcome, possibly based on values calculated during the step which has just completed. In our sample *finddups* macro we could check before the sort step whether the input data set has any observations. As another example, a macro may require a particular value be present on a data set. If the value is not present, appropriate action may be taken. See the *regspline* macro described in [8] for a concrete example.

For a limited number of procedures SAS provides the *sysinfo* macro variable. This contains return codes provided by the *compare*, *download* and *upload* procedures. In these cases, *sysinfo* provides more detailed information about the outcome in the case of success and may also be used to decide whether execution should continue or not.

9 Returning status to the caller

In most programming languages, when defining a function the language offers a way for the function to provide a return value. In a sense, the SAS macro language does this too, but the return value is essentially text which, except in the case of a pure macro, will be compiled and executed by the SAS supervisor. A solution to the problem of return codes is provided by global macro variables. The macro can place its return code in a global macro variable. The name of this variable can be provided by the caller, thus transferring the responsibility for not overwriting existing global variables from the macro to the caller. This also opens up the possibility of returning multiple values in different macro variables, as demonstrated by the *maxdups* parameter in the robust

version of *finddups* presented in Appendix A. A possible implementation of returning status is shown in the following modification of our sample macro *finddups*:

```
%macro finddups(data=_last_,out=_dups,by=, rc=rc_finddups) ;
    %if %quote(&rc) = %str() %then %do ;
        %put NOTE: &sysmacroname: Using RC_&sysmacroname for the RC parameter ;
        %let rc=rc_&sysmacroname ;
    %end ;
    %global &rc ;
    %let &rc=9999 ;

    proc sort data=&data out=&out ;
        by &by ;
    run ;
    %if &syserr>4 %then %do ;
        %let &rc=1 ;
        %return ;
    %end ;
    ...
    %let &rc=0;
%mend;
```

We use *rc_finddups* as the default value for a parameter *rc* because we will assign to the global macro variable *rc* in the macro and so the parameter *rc* must have a value. So if it has been set to missing, we reset it to the default. Further, we adopt the convention that a return code of zero indicates success. If an error occurs in the sort step we set a non-zero return code and exit. In order to eliminate the possibility of incorrectly returning zero when a problem happens which our code does not detect, we initialize the return code to a suitable non-zero value, in this case 9999. If all steps in the macro execute successfully, the return code is set to zero as the last action in the macro.

10 Restoring the environment

A macro is generally designed to carry out a specific task. Apart from changes directly related to the task, it is good practice for a macro to restore the environment to the state it was in before the macro was called. While this may not contribute directly to the robustness of the macro itself, it may do so if there are multiple calls of the macro in a single session. It also contributes to the robustness of programs which call the macro and which may call other macros afterwards.

As a general rule, any effect which is not a documented output of the macro should be undone before termination. In particular this includes resetting options to their original values, deleting temporary files and data sets, clearing file or library references used internally by the macro and ensuring that no undesired global macro variables have been defined.

While we have used the *%return* statement⁴ for premature termination in our examples so far, addition of clean-up code requires a more differentiated approach. At the point where we decide on premature termination, we must also consider whether any clean-up is necessary. For premature termination caused by failure of parameter validation, it is generally unlikely that any changes to the environment have been made and we may safely use the *%return* statement. Once we have created temporary data sets, set options, or assigned library or file references, we must use the *%goto* as illustrated in the following variation of the *finddups* macro which creates a temporary data set in the sort step:

```
proc sort data=&data out=_finddups_tmp ;
    by &by ;
run ;
%if &syserr>4 %then %do ;
    %let &rc=2 ;
    %goto DONE ;
%end ;

data &out ;
    set _finddups_tmp ;
```

⁴available only since Release 9

```

    ...
run ;
%if &syserr>4 %then %do ;
    %let &rc=3 ;
    %goto DONE ;
%end ;

%let &rc=0 ;
%DONE:
%if %sysfunc(exist(_finddups_tmp)) %then %do ;
    proc datasets lib=work nolist ;
        delete _finddups_tmp;
    quit ;
%end ;
%mend ;

```

If we were sure that a failure in the sort procedure would not leave an incomplete copy of the *tmp* data set, we could use *%return* instead of *%goto*. We also do not check the return code of the *datasets* step - a refinement would be to only check this if the macro has run successfully up to this point. If an error occurs, it is preferable to report that error.

11 Additional considerations for in-step macros

For in-step macros, no validation is likely to be the only sensible alternative. Parameters for in-step macros are most likely to be variable names, data set names, format or informat names, statement options, or lists of these. Since the in-step macro completes execution after the compilation of the step has begun, it is not possible for the caller to use the return code from the macro until after the step has run. Additionally, if the macro exits without generating any code, the enclosing step might still run without formal errors unless the macro itself writes an error message.

A further issue is the use of temporary data set variables. If a calculation requires storing an intermediate value, a temporary data set variable may be needed. Since the data set has only a single scope for variables, it is necessary to choose a name which is not used by the caller. Furthermore, the macro must ensure that the temporary variable is not written to any output data set[9]. A partial solution is to adopt the convention of naming temporary variables using a double underscore and the macro name as root and a *drop __macroname:* ; statement. Another approach which avoids the drop problem is to use temporary arrays, but this does not entirely solve the naming problem.

SAS 9.2 introduces the FCMP procedure [3] which allows a programmer to write functions for use in the data step and in selected procedures. These functions behave just like SAS-supplied functions and any variables used within the function are local to the function. If this feature is reliable it may, in time, may render in-step macros obsolete.

12 Testing

In addition to the normal testing and validation of a macro, when using the techniques described in this paper we recommend that explicit tests be made for the “robust” code. In particular, a test to prove that each parameter validation is working correctly should be done. Tests of the effects of checking outcomes are also advisable. In the case of checks on content, such as non-zero observations in a data set, this is relatively easy to do, but testing that the code behaves correctly when *syserr* is greater than 4, i.e. when an error occurs, can provide some challenges. One technique which can be used to cause a step to fail is to arrange for a data set used in the step to be inaccessible either for reading or writing. For permanent data sets this may be done by removing the appropriate access for the test user. For temporary data sets created by the macro itself, create the data set with alter and read passwords before calling the macro. For this to work for every step, the macro may not re-use temporary data set names. Alternatively, the system option *user* may be employed to have data sets with a one-level name written to a permanent library where the appropriate access restrictions may be set:

```

libname testwork "~/testwork" ;
options user=testwork ;

```

13 Documentation

Good documentation can contribute to the robustness of a macro by reducing the likelihood that the macro will be called with inappropriate parameters. In order to achieve this goal, documentation needs to cover a number of different aspects. First and foremost, a concise description of the purpose of the macro should give the user or prospective user a clear idea of what the program does and the circumstances in which it may be used, allowing the user to decide if the macro is appropriate for the task in hand. Secondly, the parameters must be well described including sufficient detail on type, expected values and default values. Possible interactions between parameters should also be addressed. Thirdly, any inputs to the macro, in particular files or data sets, should be clearly described, including specification of relevant structural or variable attributes. Fourthly, a sufficiently detailed description of the outputs of the macro should be provided, covering any type of output produced. Finally, at least one realistic example should be provided. Unless relevant to the use of the macro, the documentation should not describe the technical implementation details, although this may be provided as an appendix or, preferably, as a separate document.

We also recommend writing the user documentation before the macro is written. Since the documentation describes the user interface in detail, this may serve as the specification for the macro.

Finally, there is the question of whether to include the documentation in the code or maintain it as a separate file. Some languages such as LISP, Perl and R avoid this problem by providing features for embedding the documentation in the code and tools to produce a nicely formatted version thereof, but SAS does not provide such for macros. It is certainly useful to have at least a synopsis of the documentation covering the purpose, parameters, inputs and outputs in the macro, but is not very convenient for formatting more extensive examples. Whether documentation is included in the code or not, it is imperative to ensure that any documentation is updated when new versions of the macro are programmed. Following a software development process helps to ensure that this takes place.

14 Conclusions

The techniques described in this paper, when used appropriately, will aid in producing robust SAS macros. The benefits to be gained are improved confidence that the macros will find and report problems and therefore not produce results when none should be produced. Additionally, well-documented robust macros need significantly less support. On the cost side, there is extra effort involved in adding and testing the robust code. This effort of adding the code can be reduced to a minimum by defining templates for whatever editing tool or development environment is used⁵. Over time, implementation of the techniques becomes second nature and the cost decreases. We have found that adding the robust code after the fact is more time-consuming than applying the techniques during development.

References

- [1] SAS Institute Inc., *SAS Macro Language: Reference, First Edition*, SAS Institute Inc., 1997
- [2] SAS Institute Inc., *SAS Screen Control Language, Version 6, Second Edition*, SAS Institute Inc., 1994, pp 99-101
- [3] SAS Institute Inc., *Base SAS 9.2 Procedure Guides: The FCMP Procedure*, SAS Institute Inc., 2009, retrieved 29 March 2009 from <http://support.sas.com/documentation/cdl/en/proc/61895/HTML/default/a002890483.htm>
- [4] SAS Institute Inc., *SAS Language: Reference, First Edition*, SAS Institute Inc., 1990, pp 19-20
- [5] SAS Institute Inc., *SAS Note 4420: Condition codes are not set correctly when errors occur on the ODS statement*, SAS Institute Inc., 2009, retrieved 20 Feb 2009 from <http://support.sas.com/kb/4/420.html>
- [6] SAS Institute Inc., *SAS Note 620: Incorrect value for &SYSERR returned from Graphics procedure*, SAS Institute Inc., 1999, retrieved 20 Feb 2009 from <http://support.sas.com/kb/00/620.html>
- [7] SAS Institute Inc., *SAS Note 8522: SYSERR macro variable might generate an incorrect value and issue warning*, SAS Institute Inc., 2009, retrieved 20 Feb 2009 from <http://support.sas.com/kb/8/522.html>

⁵A set of Emacs extensions for supporting the techniques is available on request from the author

- [8] Gregory M, Ulmer H, Pfeiffer KP, Lang S, Strasak AM. *A set of SAS macros for calculating and displaying adjusted odds ratios (with confidence intervals) for continuous covariates in logistic B-spline regression models*. Comput Methods Programs Biomed. 2008 Oct;92(1):109-14.
- [9] Aster, Rick. *Professional SAS Programming Shortcuts* Breakfast Communications, 2005 p 329

Contact Information

Martin Gregory
 Head of Statistical Programming Germany
 Merck KGaA
 Frankfurterstr 250
 64293 Darmstadt
 Germany
 Martin.Gregory@merck.de

A The robust finddups macro

This appendix presents a robust implementation of the bare *finddups* macro presented in section 5. It includes two additional features. The parameter *out2* specifies a data set in which to write the unique records while the *maxdups* parameter names a global macro variable to be populated with the maximum number of duplicates for any key.

```
/*
Function: Create a data set containing duplicates of an input
         data set. This is almost the complement of PROC SORT NODUPKEY
         - it throws away all records having a unique key.

Return codes:  0: success;
               1: missing value for parameter DATA;
               2: missing value for parameter OUT;
               4: missing value for parameter BY;
               3,5,6,7: combination of parameters missing
               101: error during the sort step;
               102: error during the data step searching for duplicates.
               9999: unexpected error

*/
%macro finddups(data=_last_,      /* input data set */
               out=_dups,        /* output data set */
               out2=,            /* optional nodup output data set */
               by=,              /* by variables */
               where=,           /* where clause */
               maxdups=,         /* global macro var for max number of dups */
               rc=rc_finddups    /* global macro var for return code */
               );
  %if %quote(&rc) = %str() %then %do ;
    %put NOTE: &sysmacroname: Using RC_&sysmacroname for the RC parameter ;
    %let rc=rc_dups;
  %end ;
  %global &rc ;
  %let &rc=9999 ;
  %if %quote(&maxdups) ^= %str() %then %global &maxdups;
  %local i lastby nmiss _params _param ;
  %let i=1 ;
  %let nmiss=0 ;

  /* parameter validation */
  %let _params=.data.out.by.;
  %let i=1;
```

```

%do %while(%scan(&_params,&i,.)^=%str()) ;
    %let _param=%scan(&_params,&i,.);
    %let &_param=%nrquote(&&&_param) ;
    %if &&&_param = %str() %then %do ;
        %put ERROR: &sysmacroname: Parameter %upcase(&_param) is required.;
        %let nmiss=%eval(&nmiss+2*(&i-1)) ;
    %end;
    %let i=%eval(&i+1) ;
%end;
%if &nmiss>0 %then %do ;
    %let &rc=&nmiss ;
    %goto DONE;
%end ;

proc sort data=&data out=&out ;
    by &by ;
    %if (%quote(&where) ^= %str()) %then %do ;
        where &where ;
    %end ;
run ;
%if &syserr > 4 %then %do ;
    %put ERROR: &sysmacroname: error sorting the data set. ;
    %let &rc=101 ;
    %goto DONE;
%end;

/* determine the last by variable for use with first. and last. */
%let lastby=%scan(&by,-1,%str( ));

data &out (drop=__ndup __maxdup)
    %if %quote(&out2) ^= %str() %then %str(&out2(drop=__ndup __maxdup));
    ;
    set &out end=__end;
    by &by ;
    retain __maxdup __ndup 0 ;
    if first.&lastby + last.&lastby < 2 then do ;
        output &out ;
        if first.&lastby then __ndup=0;
        __ndup+1;
        if last.&lastby then __maxdup=max(__maxdup,__ndup);
    end;

    %if %quote(&out2) ^= %str() %then %do ;
        else output &out2 ;
    %end ;

    %if %quote(&maxdups) ^= %str() %then %do ;
        if __end then call symput("&maxdups",compress(put(__maxdup,best12.)));
    %end;

run ;
%if &syserr > 4 %then %do ;
    %put FINDDUPS: error building duplicates data set &out. ;
    %let &rc=102 ;
    %goto DONE ;
%end;

%let &rc=0 ;
%DONE:
%mend ;

```