

Simple %str(ER)ROR checking in macros

Magnus Mengelbier, Limelogic, Neuchatel, Switzerland

Custom and system SAS® macros are toolsets that are intended to contribute to daily activities. Many macros have evolved through personal or project libraries from that simple shortcut to a de facto standard. The amount of error chasing can be exceedingly cryptic and unnecessarily painful as that input parameter or data set is not just as intended. We consider very simple techniques and examples for error checks that are quick and easy to implement in any SAS macro.

INTRODUCTION

Error checking is extremely useful and adds simplicity, transparency and efficiency as the macros are more well documented, easier to implement and far simpler to debug in case of issues and errors.

The strategy and general approach is extremely important as several factors interact to provide a positive encounter. We consider error checks for macro parameters, libraries, file references, data sets and data set variables as a set of special cases of increasing complexity.

We also consider more complex checks to assert that the continued processing has all required information and is not futile. Futility checks are just as important as the simple error check. As macro and system complexity increases, terminating a task due to futility early is increasingly important and critical to system stability.

The performance degradation attributed to error and futility checks are highlighted with two general approaches. The degradation is discussed in relation to performance impact, added value and increase/decrease in efficiency.

The use of error checking in macros as presented does not consider conventions and strategy, but highlights useful techniques to consider when developing standard re-usable macro libraries.

ONE STRATEGY AND TWO APPROACHES

We consider the use of the SAS standard ERROR, WARNING and NOTE keywords as self-evident to reflect any issue or notification. Other candidate keywords, such as NOTIFICATION, MESSAGE, ALERT, SYSTEM, etc. are useful, but not as well ingrained in SAS users as the SAS messages in tri colori; the red, green and blue by default. Users, process requirements and some tools also have the SAS standard keywords embedded to an almost conditioned response.

The strategic use of the SAS keywords for errors, warnings and notes will seamlessly integrate the approach into any existing process and log scanning toolset. The intent is to facilitate and not add an extra layer, so we will strive to retain the inherent keyword association.

However, the selection of keywords should consider if it is critical to be able to discern SAS internal messages and those generated by custom and system macros. The addition of the macro name or other identification in the error, warning or note may be useful to distinguish between SAS internal messages and those generated by custom macros as well as provide an origin.

```
ERROR: This is my error message [mymacro].  
ERROR: [mymacro] This is my error message.
```

If we can obtain the same default red, green and blue colour highlighting as SAS generated messages, this would significantly add to the simplicity. In fact, the keywords ERROR, WARNING and NOTE, followed immediately by a colon, put in the log in upper case with no leading spaces or other characters, will be highlighted by SAS, just as SAS internal errors, warnings and notes.

We can therefore generate and add errors, warnings and notes to the SAS log through both SAS macro code and the DATA step. However, this approach only retains highlighting and does not set the read-only automatic macro variable SYSERR, unless an actual SAS error has occurred, which is what we are attempting to avoid in the majority of cases.

In SAS macro code, we use the %put statement to put the message in the log.

```
%put ERROR: This is my error message.;
```

The log would appropriately show a colour-highlighted error, which is our intent.

```
1    %put ERROR: This is my error message.;
```

ERROR: This is my error message.

In the event that an error has occurred, the colour-highlighted word ERROR is visible. A search for ERROR with Find, or a log checker, will also appropriately find the error message.

Unfortunately, the above convention will always find the word ERROR as part of the %put source statement, regardless of the error condition. This can be frustrating and discouraging for a user as there may not be an error, just the program logic for the error check. Too many false positives may decrease the reliance on ERROR and WARNING as signals, setting the scene for missing a true and critical message.

The use of NOSOURCE or NOSOURCE2 system option will circumvent the issue, as the %put source statement is not contributed to the log. The use of the no source system options is frowned upon by many organisations, as the entire source code is not documented in the log.

Updating the SAS macro code to use the %str() function would sufficiently mask, break up or hide the word ERROR and no longer trigger in a full word search. .

```
%put %str(ER)ROR: This is my error message.;
```

The log would appropriately show the updated %put statement and the same message.

```
2    %put %str(ER)ROR: This is my error message.;
```

ERROR: This is my error message.

The word ERROR would now only be found in the event that the error condition is true.

Similarly, we can use the same approach in a DATA step.

```
data _null_;

    put "ERROR: This is my first error message.";
    put "ER" "ROR: This is my second error message.";

run;
```

The log would naturally show

```
8
9    data _null_;
10
11        put "ERROR: This is my first error message.";
12        put "ER" "ROR: This is my second error message.";
13
14    run;

ERROR: This is my first error message.
ERROR: This is my second error message.
NOTE: DATA statement used:
      real time           0.00 seconds
      cpu time            0.00 seconds
```

Similar to the macro example, a search on the keyword ERROR will find the put "ERROR: " statement for the first message as well as the error message, while the second syntax put "ER" "ROR: " will only find the triggered error message. The approach for WARNING is appropriately identical.

For the duration of the paper, we will for brevity discuss error and futility checks as macro code, but the examples are equally applicable in DATA steps, provided slight syntax changes.

SAS MACRO ANATOMY

There are a many different approaches to the design of custom and system macros. One approach is to optimise macro execution by only executing the entire macro when all, or at least the critical conditions, have been satisfied.

The first set of checks to complete will verify, or assert, that all the necessary parameters have a value, a value within a range or a value from a given list of keywords. As checks are most often rudimentary and designed to capture obvious conditions without any complex processing, they logically occur at the very start of the macro prior to any further and expensive processing. If one or more of these checks fail, the macro will exit with appropriate messages. As more than one condition may fail, it would be sensible to perform as many checks as possible before exiting.

A simple approach is to employ a macro flag variable, say *sighup*, to indicate if at least one error check has failed.

```
%local sighup ;

%let sighup = 0;    /* - default is no error ;

%if ( ... ) %then %do;
    %put %str(ER)ROR: My first parameter has an error. ;
    %let sighup = 1;
%end;

%if ( ... ) %then %do;
    %put %str(ER)ROR: My second parameter has an error. ;
    %let sighup = 1;
%end;

%if ( &sighup > 0 ) %then ... ;    /* - error has occurred, then exit ;
```

Similarly, the flag variable can be defined from within a DATA step using the call symput() statement.

```
data _null_ ;

    /* - some data step code ;

    call symput('sighup', '1');
    call symput('sighup', put(error_count, best.-L) );

    call symput('sighup', 1);

run;
```

All three statements will assign a value to the flag variable. The third statement will generate an unnecessary numeric-to-character conversion message (below) as well, which can sometimes be used to indicate un-initialised or missing variables. Also note the left alignment (-L) associated with the format in the put statement, which will avoid leading spaces.

```
2    data _null_ ;
3
4        /* - some data step code ;
5
6        call symput('sighup', '1');
7        call symput('sighup', put(error_count, best.-L) );
8
9        call symput('sighup', 1);
10
11    run;
```

```
NOTE: Numeric values have been converted to character values at the places given by:
(Line):(Column).
      9:26
```

The second set, futility checks, would verify that conditions are sufficient for continued processing. Futility checks usually require some degree of processing and are intended to capture situations where the initial error checks lack sufficient information or prior processing to trigger errors or warnings. The border between error and futility checks is not clearly defined, but more based on conjecture and the relative position in the macro program code. Attempts to define the border between error and futility checks usually consider the degree of macro logic processing prior to a specific check, e.g. none or any processing. Advanced macros may also use futility checks to adopt default values for continued processing based on more complex relationships or conditions.

Futility checks have two powerful aspects. A futility check will terminate the execution of the macro as soon as a terminal condition is encountered. The futility check is also not bound to a specific location within a macro anatomy, but can be used as execution progresses. For example, the first futility check within a macro may verify that a set of specified variables exist, while further futility checks farther into the macro would highlight any issues with a critical merge, such as more than one record per a predefined key sequence.

%if - %then - %else or %goto

There are two approaches to controlling macro execution while performing error checks. The first, and sometimes considered more cumbersome, method would be large and nested %if - %then - %else blocks. This approach is easy to complete if the macro is extremely short and only one or two checks are performed. Larger blocks straddling a greater number of lines of code may not be easy to read and manage.

We consider the %goto statement and associate label an applicable surrogate for the nested %if - %then - %else blocks, if the %goto statement is used constructively.

```
%if () %then %goto foo;

%* - first block of sas code ... ;

%if () %then %goto bar;

%* - second block of sas code ... ;

%bar:

%* - third block of sas code ... ;

%foo:

%* - fourth block of sas code ... ;
```

The above example illustrates a simple approach to manage the blocks of macro code to be executed. Note that the jumps as depicted show skips further down the macro, even though skips back towards the beginning are possible. The convention of allowing looping and random skips can severely reduce readability, clarity of progress and be more difficult to trace [1].

By convention, we use the label *exit* as an appropriate label to indicate an exit point. A previous example would therefore be more complete as below.

```
%local sighup ;

%let sighup = 0;    %* - default is no error ;

%if () %then %do;
    %put %str(ER)ROR: My first parameter has an error. ;
    %let sighup = 1;
%end;

%if () %then %do;
    %put %str(ER)ROR: My second parameter has an error. ;
    %let sighup = 1;
%end;
```

```

%if ( &sigmap > 0 ) %then %goto exit ;  %* - error has occurred, then exit ;

%* - macro code to be executed if not obvious errors are found ... ;

%exit:

%* - macro code to be executed on exit ... ;

```

Both approaches are appropriate, however the %goto method may very well provide a more legible code, if used in a controlled and consistent manner. The succeeding examples are all based on the %goto method, but the approach using %if - %then - %else blocks should be functionally equivalent. The %if - %then - %else blocks will also require a more constructive and considerate macro design, such that the nesting is reduced to a minimum.

DATA STEP ANATOMY AND THE RETURN

Similar to the macro anatomy and the %goto statement, we can perform error and utility checks in a DATA step. The RETURN statement without optional labels is extremely useful, as the encounter of a RETURN statement will stop processing the current record and return to the top of the DATA step.

For example, the utility check that all date parts (numeric variables DD, MM, YYYY) are non-missing prior to generating a date value is extremely simple.

```

data work.out;
  set work.dates ;

  format thedate date9.;

  if ( nmiss( dd, mm, yyyy ) gt 0 ) then do;
    put "ER" "ROR: The expected date is incomplete.";
    return;
  end;

  thedate = mdy(mm, dd, yyyy);
run;

```

A more powerful example is to manage unexpected records. Program processing can be unpredictable, if multiple records are encountered when one is expected. For example, if one record per subject is expected, the below code would clearly document the exception of more than one record present.

```

data work.out;
  set work.in;
  by subject ;

  if first.subject and (not last.subject) then do;
    put "ER" "ROR: More than one record encountered for subject " subject ".";
    return;
  end;

  * - data step code to be executed.;
run;

```

For example, if a structure or control data set is only to include one record for a given selection, we can use the RETURN statement to control the execution.

```

data work.out;
  set work.structure;

```

```

where ( ... );

if ( _n_ > 1 ) then do;
    put "ER" "ROR: More than the expected one record encountered in ...";
    return;
end;

* - data step code to be executed.;
run;

```

These approaches are also very applicable and powerful to all programming in general.

CHECK FOR A VALUE

The simplest of checks is verifying that an arbitrary value has been provided.

```

%if ( %str(&foo) eq %str() ) %then %do;
    %put %str(ER)ROR: Please specify a value for foo.;
    %goto exit;
%end;

```

The use of the %str() function serves two purposes. The initial purpose is to mask special or mnemonic characters during compile time, such that interpretation is performed during macro execution [2]. The second use, to the right of the equal operator *eq*, is for readability. The intent of the if-statement is much more obvious, if we state *%if (%str(&foo) eq %str())* than the shorter *%if (%str(&foo) eq)*, or possibly *%if &foo eq %then*, which can be confusing at times.

Another approach would make use of the %length() function.

```

%if ( %length(&foo) eq 0 ) %then %do;
    %put %str(ER)ROR: Please specify a value for foo.;
    %goto exit;
%end;

```

Care has to be taken with characters representing arithmetic operations, as these may be evaluated as numeric operators. The condition of the *%if (condition)* statement is equivalent to the *%eval(condition)* statement, resulting in a numeric arithmetic evaluation at some point in assessing the *condition*, if the condition can be interpreted as an arithmetic expression.

Additional consideration may have to be provided to the possible structures of the parameter values. For example, a parameter that may contain a large number, say the unique subject number 95642239112, may provide unexpected results that can be misleading.

```

%macro test(subject = );

    %if ( %str(&subject) = %str() ) %then
        %put Subject is empty.;

%mend;

%test( subject = 0 );

%test( subject = 95642239112 );

```

The result is surprising error condition not obvious to the location, even less in a very large macro.

```

12 %macro test(subject = );
13
14     %if ( %str(&subject) = %str() ) %then
15         %put Subject is empty.;
16
17 %mend;

```

```

18
19
20 %test( subject = 0 );
21
22 %test( subject = 95642239112);
ERROR: Overflow has occurred; evaluation is terminated.
ERROR: The macro TEST will stop executing.

```

The error occurs as SAS attempts to interpret the string of characters 95642239112 as a number, which is too large for the given SAS version and platform (example from SAS 8.2 on Microsoft Windows XP). There are several approaches to circumvent this issue; the most uncomplicated is to precede the value of subject with a character prefix, such that the value is interpreted as a character sequence. We select the prefix `'_a'` (underscore-a).

```

%if ( %str(a_&subject) = %str(a_) ) %then
  %put Subject is empty.;

```

The prefix `"a_"` forces the sequence `a_95642239112` to be interpreted as a text value, rather than the original numeric. The above example provides an indication that understanding possible parameter values is critical in order to avoid introducing new, unexpected and complex error messages.

Check for a numeric value

As any value may be too broad for some macro parameters, we can carefully extend the previous check to assert numeric arguments, provided that the expected values are not too large. This can fairly simply be accomplished using the `compress()` function. The verbose syntax is provided, which can be easily extended and enhanced with `compress()` function modifiers (third function argument).[3]

```

%if ( %sysfunc(compress( &foo, 0123456789 )) ne %str() ) %then %do;
  %put %str(ERROR: The parameter foo is not an integer. ;
  %goto exit;
%end;

```

The above check considers whole numbers only. As some parameters may allow decimals, negative numbers or scientific notation, additional symbols may be added to the list of characters in the second argument for the `compress` function along with more complex logic to check the position of the decimal character, minus sign, etc.

SAS version 9 introduces the `notdigit()` function, which is equivalent to the above verbose example and a little less flexible as it cannot manage negative and decimal values, as the function performs a strict check for any character not a digit from 0 to 9. Full flexibility in SAS version 9 can be accomplished using Perl regular expressions with the `rxparse()` and `rxmatch()` functions, to verify that the item is in the form of a number.

CHECK FOR A KEYWORD

Flags and keywords are useful to control execution and expected macro behaviour. The effort to verify that the value provided is acceptable can sometimes be difficult. A simple approach using `indexw()` function can reduce the daunting task to a simple exercise.

For example, consider the simple option `debug`, which should take values Yes, No and the first letters Y and N as abbreviations.

```

%if ( %sysfunc(indexw( %str(y yes n no), %sysfunc(lowercase(&debug))) ) = 0 ) %then %do;
  %put %str(ERROR: The debug option %str(&debug) is not recognized.;
  %put %str(ERROR: Valid values for the debug option are Yes, Y, No and N.;
  %goto exit;
%end;

```

The check verifies the `debug` option as case insensitive using lower case. All the appropriate values are provided as the first argument to the `indexw()` function.

The list of keywords may also be contained in a data set for more complex cases. Consider as an example, the list of summary tables contained in the data set `list_of_tables`.

```
%local table_exist ;

%let table_exist = UNKNOWN;

proc sql noprint;
    select coalesce( table_ref, "MISSING") into: table_exist separated by " "
        from list_of_tables
        where ( upcase(trim(left( table_id ))) = upcase(trim(left( "&table" ))) )
    ;
quit;

%if ( %sysfunc(lowcase(&table_exist)) eq %str(unknown) ) %then %do;
    %put %str(ER)ROR: The summary table %str(&table) is unknown.;
    %goto exit;
%end;
```

There are several aspects of this check that are interesting to discuss, even though the approach can be argued suboptimal.

- The local macro variable `table_exist` is initialised with the keyword value `UNKNOWN`. The keyword was selected as to indicate that the table ID is unknown, and not directly an error within the scope of this check. The macro variable `table_exist` will only retain the value `UNKNOWN`, if the SQL statement does not return any rows.
- The SQL-statement makes use of the `coalesce()` function to assign a value `MISSING` in case `table_ref` is missing for an existing `table_id`. If you consider a data set record with a missing table ID as equivalent to an entire missing record, selecting the keyword `UNKNOWN` instead of `MISSING` would be appropriate.
- The use of *separated by* “ ” is only retained to capture multiple and not just the last returned record. Obtaining a space-delimited list of table references may indicate a duplicate table ID.

In principle, this check contains more processing than the simple checks we have considered previously. This would indicate that the check might be more appropriate as a futility check, rather than an error check.

CHECKS FOR A LIBRARY

Macros that use one or more libraries as input are useful. Unfortunately, the error messages for unassigned or incorrectly configured libraries are very much dependent on the procedures and other constructs they may use.

The error check to ascertain if a library is assigned is extremely straightforward.

```
%if ( %sysfunc(libref( &foo )) ne 0 ) %then %do;
    %put %str(ER)ROR: Input library %str(&foo) is not correctly defined.;
    %goto exit;
%end;
```

The `libref()` function will return a value other than 0, if there is an error with the assigned library reference. For example, assign a library names `foo` and `bar` to a path that does not exist.

```
libname foo "/foo/bar";
libname bar "/foo/bar";
```

The result is an assigned library reference. However, the fact that the path does not exist is only mentioned as a SAS note when we assigned `foo` and a warning when we assign `bar`, even though we assign the library to the same physical path.

```
137 libname foo "/foo/bar";
NOTE: Library F00 does not exist.

138 libname bar "/foo/bar";
NOTE: Libname BAR refers to the same physical library as F00.
WARNING: Library BAR does not exist.
```


NOTE: Libref BAR was successfully assigned as follows:

Engine: V8
Physical Name: C:\foo\bar

Note that the SAS note associated with the library *bar* assignment can be misleading as well, adding an additional argument to perform library error checks.

There is a simple way to include path in the scope of a library check. The `fileexist()` function is useful to verify that a physical path exists for an assigned library, provided one is to be assigned. Note the nested calls with the `%sysfunc()` functions.

```
%if ( %sysfunc(fileexist( %sysfunc(pathname( &foo )) )) eq 0 ) %then %do;  
  %put %str(ER)ROR: Input library %str(&foo) path does not exist.;  
  %goto exit;  
%end;
```

The same process can be applied to file references using the `fileref()` function, as opposed to the `libref()` function.

Complexity increases as you wish to add further error checks on libraries assigned to databases, such as Oracle, or other similar sources, which can require an attempt to select data or information through the connection.

CHECK FOR A DATA SET

Eventually a macro will use an input data set, either as a source of information or a control mechanism similar to that of PROC FORMAT and the `cntl` attribute. Quite often, a common usage issue is that either an expected data set does not exist or, possibly, one exists where it is not expected. The latter error check can be extremely useful in protecting existing output data sets.

The most simple mechanism is to use the `exist()` function, which our example restricts to check for the existence of a single data set.

```
%if ( %sysfunc(exist( &foo , DATA )) eq 0 ) %then %do;  
  %put %str(ER)ROR: Data set %str(&foo) does not exist.;  
  %goto exit;  
%end;
```

The `exist()` function can also verify the existence of a SAS access definition, views and catalogs. If the DATA option is not specified, SAS will look for any of the mentioned types and may return a false positive.

If the library is not known and the mere existence of a data set *foo* in any library will suffice, the use of `DICTIONARY.TABLES` is just as applicable.

```
%local dataset_exist ;  
  
%let dataset_exist = ;  
  
proc sql noprint;  
  select distinct(memname) into: dataset_exist separated by " "  
    from dictionary.tables  
    where ( upcase(trim(left( memname ))) = upcase(trim(left( "&foo" ))) )  
  ;  
quit;  
  
%if ( %str(&dataset_exist) eq %str( ) ) %then %do;  
  %put %str(ER)ROR: The data set %str(&foo) does not exist.;  
  %goto exit;  
%end;
```

This above check is sufficient, if the library is not required to be specified. The above selection will return a space-delimited list of every occurrence of *foo*, disregarding the library association.

More information can be gained by extending the check to include the library name in the space delimited list *datasets* as below.

```

%local dataset_exist ;

%let dataset_exist = ;

proc sql noprint;
    select compress(libname || "." || memname, " ") into: datasets separated by " "
        from dictionary.tables
        where ( upcase(trim(left( memname ))) = upcase(trim(left( "&foo" ))) )
    ;
quit;

```

A prudent approach would be to consider and manage the occurrence of a library in association with the specified data set. If the data set *foo* is specified with a library, case specific parsing may have to be performed. This is easily implemented with the %index() and %scan() functions. We choose to add the default WORK library reference instead of constructing a condition around the PROC SQL where clause.

```

%local dataset_exist
    full_foo
    ;

%if ( %index( &foo, %str(.) ) > 0 ) %then %let full_foo = &foo ;
    %else %let full_foo = WORK.%str(&foo) ;

%* - some macro code ... ;

%let dataset_exist = ;

proc sql noprint;
    select memname into: dataset_exist separated by " "
        from dictionary.tables
        where ( upcase(trim(left( libname ))) =
            "%upcase( %scan( &full_foo, 1, %str(.) )" ) and
            ( upcase(trim(left( memname ))) =
                "%upcase( %scan( &full_foo, 2, %str(.) )" )
            )
    ;
quit;

%if ( %str(&dataset_exist) eq %str(.) ) %then %do;
    %put %str(ER)ROR: The data set %str(&foo) does not exist.;
    %goto exit;
%end;

```

The check code documents the default location to the WORK library, which may, or may not, be the desired effect. The convention enables the default location to be any process library, and not just WORK.

CHECK FOR A VARIABLE IN A DATA SET

A method to check for one or more missing variables in a specified data set is similar to that of checking the existence of a data set with SAS system dictionaries. The Dictionary view COLUMNS is used instead of the previous TABLES, as this provides information about column attributes.

We shall make use of three local macro variables.

- *data_full* contains the fully qualified data set name, including the library reference.
- *variables_quoted* is each element of a space delimited list of variable names quoted, except the first and last item that has unbalanced quotes. Note that the full list of elements has balanced quotes, since the missing first and last cancel each other out.
- *missing_variables* will contain a space delimited list of any missing variables names.

For our example, we assume the macro parameter *variables* is a space delimited selection of variables to process from the input data set *data*.

```

%local data_full
    variables_quoted
    missing_variables
    ;

```

```

%if ( %index( &data, %str(.) ) > 0 ) %then %let data_full = &data ;
                                %else %let data_full = WORK.%str(&data) ;

%* - create simple list of selected variables ;
%let variables_quoted = %sysfunc(tranwrd( %str(&variables), %str( ), %str(", ")));

data work.__selected_var__ ;

    length varname $ 50 ;

    do varname = "&variables_quoted" ;
        output;
    end;
run;

%* - generate list of missing variables ;
%let missing_variables = ;

proc sql noprint;
    create table work.__dataset_var__ as
        select name from dictionary.columns
            where ( upcase(trim(left( libname ))) =
                    "%upcase( %scan( &data_full, 1, %str(.) )" ) and
                  ( upcase(trim(left( memname ))) =
                    "%upcase( %scan( &data_full, 2, %str(.) )" )
                )
            ;

    select varname into: missing_variables separated by " "
        from work.__selected_var__
        where ( upcase(trim(left( varname ))) not in
                (select distinct(upcase(trim(left(name)))) from work.__dataset_var__)
            )
;
quit;

%if ( %str(&missing_variables) ne %str() ) %then %do;
    %put %str(ER)ROR: Selected variables not found in the data set %upcase(&data).;
    %put %str(ER)ROR: Missing variables : %upcase(&missing_variables) ;

    %goto exit;
%end;

```

PROC CONTENTS and the Dictionary table COLUMNS are interchangeable in our context. Similarly, the DATA and PROC SQL steps can be replaced with macro code to compare two space delimited lists of variables names.

```

%local data_full
        i
        str
        existing_variables
        missing_variables
;

%if ( %index( &data, %str(.) ) > 0 ) %then %let data_full = &data ;
                                %else %let data_full = WORK.%str(&data) ;

%let missing_variables = ;

proc sql noprint;
    select name into: existing_variables separated by " "
        from dictionary.columns
            where ( upcase(trim(left( libname ))) =
                    "%upcase( %scan( &data_full, 1, %str(.) )" ) and
                  ( upcase(trim(left( memname ))) =

```

```

                                "%upcase( %scan( &data_full, 2, %str(.) ) )" )
;
quit;

%let i = 1;
%let str = %scan( &variables, &i, %str( ) );

%do %while ( %str(&str) ne %str() );

    %if ( %sysfunc( indexw(&existing_variables, &str) ) = 0 ) %then
        %let missing_variables = &missing_variables &str ;

    %let i = %eval( &i + 1 );
    %let str = %scan( &variables, &i, %str( ) );
%end;

%if ( %str(&missing_variables) ne %str() ) %then %do;
    %put %str(ER)ROR: One or more selected variables not found in the data set
%upcase(&data).;
    %put %str(ER)ROR: The following variables are missing: %upcase(&missing_variables);
    %goto exit;
%end;

```

The use of space delimited lists and the method of comparison may not be preferred as name literals containing spaces or other characters with dual meaning are not easy to manage consistently without great effort.

The %exit:

The termination of a macro, either due to the completion of tasks or on errors, is a critical step. An unmanaged exit can contribute to incorrect and unpredictable behaviour, poor performance and inconsistent errors. The exit is sometimes overlooked, but can be critical for a stable set of tools and/or system.

Convention and overall macro requirements would dictate the tasks to perform on exit. Complex macros can in rare cases have more than one entry into the exit routine depending on how far macro execution has progressed, but this eventuality can simply be avoided with a good macro design.

A good convention is for the exit step to return the system to the same state, which was found at invocation, with the caveat that any macro output or task should be retained accordingly. This includes system options, libraries, file references and work areas.

Macros may require specific system options to be configured for processing. A common example is SAS system options XWAIT | NOXWAIT and XSYNC | NOXSYNC.

```

%* - system option states are store in
    macro variables with prefix option_ ;
%local option_xwait option_xsync;

%* - get option settings at invocation ;
%let option_xwait = %sysfunc(getoption( xwait ));
%let option_xsync = %sysfunc(getoption( xsync ));

options ... ;

%* - macro does something ... ;

%exit:

%* - get option settings on exit ;
options &option_xwait &options_xsync ;

```

Macros may also require specific temporary libraries for processing, configuration and other functions. Proactively managing library assignments may be a validation requirement or an approach that just adds a sense of stability.

```
%* - create library ;
libname temp001 "/foo/bar";

%* - macro does something ... ;

%exit:

%* - clear library on exit ;
%if ( %sysfunc(libref( temp001 )) = 0 ) %then %do;
    libname temp001 clear;
%end;
```

Proactive management of file references follow the same argument as libraries. Certain computing environment configurations also require the program to explicitly clear a file reference assignment or release the file in order for the file to be accessible to other applications and not remain locked by the program process. This issue is increasingly observed in systems with automatic or semi-automatic version control. In order to manage file references, we use an analogous approach to that of libraries.

Active management of resources are also extended to data sets. The WORK library is an obvious location for macro specific temporary data sets, catalogs and other files, both by design and convention. A few organisations have standards that advocate a macro specific subdirectory under the WORK library to provide an isolated environment. Regardless of the convention, it is good practice to remove temporary files on exit, providing a clean environment if the macro would be executed again.

A simple convention is to apply a macro specific prefix for temporary data sets. This simplifies data set naming conventions, debugging and also removing temporary files on exit. The macro %test() in the standard toolkit has the prefix `__tk_test_` in the data set naming convention.

```
data work.__tk_test_varlist__ ;
    set work.__tk_test_contents__ ;

%* - some data set logic ;

run;

%exit:

%* - remove temporary data sets on exit ;

proc datasets library = work nodetails nolist;
    delete __tk_test_ ; run;
quit;
```

The PROC DATASETS delete method used with the colon operator makes removing temporary data sets from the WORK library a very straightforward exercise, provided the prefix is unique to the data sets in question.

FURTHER INFORMATION AND SYMSG() FUNCTION

The cases considered through the preceding error and utility checks are proactive in nature, constructed to avoid SAS errors and warnings. Eventually, a check that a statement or function has executed without errors or warnings is necessary. The SYSERR macro variable and sysmsg() function can provide useful information, although the severity, e.g. ERROR, WARNING or NOTE, reported would not be under the control of the macro developer.

```
%if ( %sysfunc(libref( &foo )) ne 0 ) %then %do;
    %put %sysfunc(sysmsg());
```

```

    %put %str(ER)ROR: Input library %str(&foo) is not correctly defined.;
    %goto exit;
%end;

```

If the library is not assigned, the sysmsg() function returns an error. The second ERROR statement is our custom message.

```

ERROR: Libname TEST is not assigned.
ERROR: Input library test is not correctly defined.

```

If the library path does not exist as in our previous example, the sysmsg() function returns a WARNING.

```

WARNING: Library FOO does not exist.
ERROR: Input library foo is not correctly defined.

```

Further descriptive information can be obtained as to why the library does not exist through additional error or futility checks.

ERROR AND FUTILITY CHECKS VERSUS PERFORMANCE

Error and futility checks in custom and standard macros are extremely practical. The collection adds simplicity, transparency and efficiency as the macros are more well documented, easier to implement and far simpler to debug in case of issues and errors. Error and futility checks also consume performance to a varying degree as processing resources are delegated to checks rather than the primary processing.

Macros are, in a sense, configurable code elements. Beyond all macro development, testing, validation, deployment, etc. they can be said to progress through three simplistic stages in implementation; development, configuration and execution. These stages also require a varying degree of interaction and error checking as well as user feedback and system integration.

A SAS macro buried deep within a system under protected and extremely well controlled conditions may not require as extensive error checking during the execution stage as a toolkit macro for one-off programs with constant updates. Similar to processing design, the relevance and level of error and futility checks are considerations for the design phase.

The *always-on* model is very common as this can provide a high level of protection. The error and futility checks are assumed a part of the macro's primary processing. A debug option is frequently available that enables the end-user to suppress cleaning the work environment. This will greatly facilitate error resolution as any temporary and macro specific libraries, file references, data sets, etc are retained and not cleared or removed as under normal operation.

```

%exit:

%if ( %sysfunc(indexw( %str(y yes), %sysfunc(lowercase(&debug))) ) > 0 ) %then
    %goto debug_selected ;

%* - cleaning steps when no debug is selected ...;

%debug_selected:

```

High performance systems will most often implement the converse. The processing time allocated to error checking can be perceived as a critical performance degradation difficult to justify. The debug option could therefore activate non-critical error and futility checks as well as retaining any temporary elements. This would in theory advocate error and futility checks, while retaining as much performance as possible.

There are certain error and futility checks we have discussed that are known for poor performance. Checks that rely on the Dictionary tables and SASHELP views are two examples of SAS resources, which consume palpable performance. The mentioned tables and views are extremely helpful and critical to some checks; therefore calling them once is better than none at all.

The error and futility checks should also be weighed in relation to using other SAS standard tools for debugging. Most organisations view error and futility checks as user level checks, while the more verbose, and at times cryptic, MPRINT, MLOGIC, and SYMBOLGEN are developer tools to trace macro execution.

CONCLUSION

Error and futility checks are extremely useful and add simplicity, transparency and efficiency as the macros are more well documented, easier to implement and far simpler to debug in case of issues and errors.

The examples we have discussed are simple techniques and cases for checks that are quick and easy to implement in any SAS macro. The checks are also applicable for high performance systems as well to facilitate and simplify debugging.

The choice of keywords, message content and the degree of error checking are all design considerations that can provide a more user friendly programming environment.

REFERENCES

- [1] Go To Statement Considered Harmful, Dijkstra, *ACM, Vol. 11, No. 3, March 1968, pp. 147-148.*
- [2] Using the %STR and %NRSTR Functions, *SAS Macro Language Reference*
- [3] compress() function, *SAS Language Reference: Dictionary*

ACKNOWLEDGEMENTS

Jan Skowronski, Genmab A/S, and David Smith, SAS Institute UK for their review time.