

Implementing User-Friendly Macro Systems

Ed Heaton – Data & Analytic Solutions, Inc.

Sarah Woodruff – Westat

SAS® facilitates macro libraries through the autocall facility and compiled macro catalogs. These permit ease of use for a set of macros with a project or corporate-wide scope. However, simply dropping a macro into an autocall library or compiling it to a catalog does not make it user-friendly. And if it is not user-friendly, it is less likely to be used.

This paper discusses robustness, reusability, and encapsulation (freedom from concern about macro implementation details) as keys to success. The paper also describes the implementation of "help" commands to enhance usefulness of the macro library. Further discussion includes macro facility options: autocall libraries as code files in system folders or as members of SAS catalogs and compiled macro catalogs.

The Problem

Many of us like macros that we use in our jobs. Often we realize that those macros are useful for other people and we can share those through a SAS autocall library. It is relatively easy to set up that autocall library. However, there should be more strict standards for macros that will be used by others – standards such as reusability, robustness, and encapsulation.

What is an autocall library?

An autocall library is simply a collection of text files in a directory. Those text files must have a **.sas** extension and the name of the text file must be the name of the macro that is called. We will call this the primary macro. The text file can contain more than one macro, and all those macros will be compiled into a macro catalog called **sasmacr.sas7bcat** in the **work** library when we call the primary macro. For instance, if we have a file called **foo.sas** and in that file are definitions for macros **%foo()**, **%getData()**, and **%writeReport()**, all these macros will be compiled when we call **%foo()**, but only **%foo()** will be executed unless **%foo()** calls one of the others.

Certain system options are required in order to effectively use an autocall library.

- **mAutocall** allows SAS to use an autocall library
- **sasAutos=()** specifies the autocall libraries and the order for macro searches among those libraries
- **mAutosourceLoc** asks SAS to write to the log the fully qualified name of the macro file whenever an autocall macro is called
- **mPrint** writes the fully resolved SAS code from autocall macros to the SAS log

Issues when others use these macros

If we write a macro in a SAS program because e.g. we want to run the same code more than once or we want to use macro looping for conditional statements which can only be used in a macro, that macro can be written to simply do the job we want it to do. However, if we write a macro that other people are going to use, we need to think about the different nuances of the task they might want it to do. We need

to do that in a way that does not require those users to look at the source code and pre-process their data to conform to the requirements of the macro.

Consider the following `%getOrderedVarNames()` macro. The user must know that it writes all of the variables from the input data set. They can tell that by looking at the code, but we should not expect the user to look at the macro code in order to understand how to use it. If we put this macro – however useful – in an autocall library it will probably not be used. We will show how to turn this macro into something more generally useful.

```
%macro getOrderedVarNames(data= ) ;
  %local datasetId ;
  %let datasetId = %sysFunc(
    open( %scan( &data , 1 , %str(%) ) , i )
  ) ;
  %local i ;
  %do i=1 %to %sysFunc( attrN( &datasetId , nVars ) ) ;
    %local oneVar varList ;
    %let oneVar = %sysFunc( varName( &datasetId , &i ) ) ;
    %if %length( %str(&varList) )
      %then %let varList = &varList &oneVar ;
      %else %let varList = &oneVar ;
    %end ;
    %let datasetId = %sysFunc( close(&datasetId) ) ;
    &varList
  %mEnd getOrderedVarNames ;
```

Goal: Create a user-friendly structure so others want to use the macro

It will be useful to define some terms.

Encapsulation

Encapsulation is the principle of information hiding – specifically hiding design decisions and details. This enables the developer to change or enhance the macro without affecting its use. It relieves the user of the burden of knowing how the macro accomplishes its task. This is not always simple to achieve. If we don't require the user to jump through hoops to set up a condition before the macro is called then the macro itself must jump through hoops to make sure it sets up the conditions in order to run properly when the user makes seemingly logical assumptions. For example, if the macro is going to write a file to a directory and that file will only be used by the macro then the macro should guarantee that the directory exists rather than expecting the user to know that it is going to write to a directory, know the name of the directory, and confirm that it exists. This level of detail needs to be hidden from the users; they should not have to worry about such implementation details.

Robustness and reusability

To be robust a macro needs to check its input, make adjustments if needed where it can, and exit gracefully if the input cannot be adjusted to conform to the requirements of the macro. By "exit gracefully", we mean the macro should report the problem to the user – usually to the SAS log – and possibly change the SAS environment to minimize the impact of the macro's failure. (E.g., set the **noReplac**e system option)

Suppose we have a Boolean parameter, e.g. **quoted=**. This Boolean parameter is used to ask whether a returned list should be quoted or not. We could expect values of **1** and **0** as being logical, but what if the user passed **quoted=yes**? Or even **quoted=y**? Do we want to tell the user to clean up their act or should we upcase the first character and see if it is either a **1** or a **Y** and if so, reset it to **1**? Wouldn't this make the macro more user-friendly and thus more robust? What if they passed **quoted=sure**? There are limits so maybe we should write a message to the log saying the macro expected **1** or **0** and then either set an error flag or **obs=0** or whatever else might be appropriate given the needs of the macro.

Example

Let's look at the macro used to write a variable list. It served our purpose well. We wanted it to create an ordered list of variables in a SAS dataset that we used in a retain statement and a keep statement in a data step. Now we want other people to use the macro, but maybe they want to use it in **Proc sql**? Now the list needs to be comma delimited. Or maybe they want to use it in a hash object definition. Now it needs to be both comma delimited and quoted. What if they don't want to use all the variables? We need to let them pass a list of variables to leave out. These considerations will determine the reusability of our macro.

Let's enhance this macro.

We have a production job for the Bandit project that writes monthly reports. One of our programmers became annoyed because she would have to create a directory for the reports before she ran the job or the job would bomb with "directory not found". So she wrote this little macro to have the program create the directory. She then calls this macro passing **date=today()** and her problem was solved.

```
%macro addMonthFolder( date= ) ;
  %sysFunc( dCreate(
    %sysFunc( putN( &date , yymmN6. ) )
    , \\server1\Bandit\Reports
  ) )
%mEnd addMonthFolder ;
```

She showed us what she wrote. We thought it was a great idea and suggested that it would be useful for other production jobs. However, some things obviously need to change to make it more generally useful.

Corporate macros

The first thing we need is an autocall library that is accessible to all the SAS users in our corporation. In our environment, that is simply a directory on one of our Windows servers. We chose

\\server2\SAS\macroLib. Our company already has this autocall library so we simply need to move our macro to that location. Later we will discuss how to set up such an autocall library.

Making your autocall library accessible

SAS comes with autocall libraries. These are defined in the configuration file – e.g.,

C:\Program Files\SAS\SASFoundation\9.2\nls\en\SASV9.CFG.

```
/* Setup the SAS System autocall library definition */
-SET SASAUTOS (
    "!sasroot\core\sasmacro"
    "!sasext0\assist\sasmacro"
    "!sasext0\eis\sasmacro"
    "!sasext0\ets\sasmacro"
    "!sasext0\genetics\sasmacro"
    "!sasext0\graph\sasmacro"
    "!sasext0\iml\sasmacro"
    "!sasext0\or\sasmacro"
    "!sasext0\qc\sasmacro"
    "!sasext0\stat\sasmacro"
)
```

We should probably not change this code. You should have another configuration file – something like **C:\Program Files\SAS\SASFoundation\9.2\SASV9.CFG**. It probably contains just one line to point to the configuration file above.

```
-CONFIG "C:\Program Files\SAS\SASFoundation\9.2\nls\en\SASV9.CFG"
```

Let's put our changes here, after SAS returns from the configuration file under the **nls** (native language support) directory.

```
-CONFIG "C:\Program Files\SAS\SASFoundation\9.2\nls\en\SASV9.CFG"
-sasAutos sasAutos "\\server2\SAS\macroLib"
```

Our task is now to distribute this configuration file to all our corporate users. But this can be a maintenance problem if we want to change this configuration file. So, let's instead create a configuration file with the **-sasAutos** option on our network and change the above configuration file to point to that network configuration file.

```
-CONFIG "C:\Program Files\SAS\SASFoundation\9.2\nls\en\SASV9.CFG"
-config "\\server2\SAS\networkConfigFile\SASV9.CFG"
```

Now we can change the network configuration file without touching the users' workstations.

But, what happens when our users want to add a project-specific autocall library? If they code

```
Options sasAutos=( "\\server42\Bandit\macros\" sasAutos )
```

they will lose the corporate autocall library. So, they need to capture the current **sasAutos** value and add to it.

```
Options sasAutos=(
    "\\server42\Bandit\macros\"
    %sysFunc( getOption(sasAutos) )
) ;
```

Macro design

Macros need headers

Any program needs a header to describe what the program does and give such details as who is responsible for the development and maintenance of the program, how it is used and a history of modifications. We use a header such as follows:

```

/*****
MACRO: %assuredPath()

OBJECTIVE:
    This macro will return a path.  If the directory does not exist,
    it will be created.

PREREQUIRMENTS:
    LibName macroCat "\\server\MacroStore\" access=readOnly ;
    Options mStored sasMStore=macroCat mAutoSource ;

PROGRAMMER:
    Ed Heaton, eheaton@dasconsultants.com

STORAGE:
    This source code is kept in
    \\server2\sas\macroLib\assuredPath.sas.

AUDIT TRAIL:
    20080611 EH Developed macro to return a valid path, even if part
                of the tree does not already exist.  It will create
                the missing parts.
    20081104 EH Added a positional parameter for help.
    20081124 EH Replaced some %PUT statements with calls to the
                %message() macro.
*****/
```

Descriptive macro names

The name of a macro, its parameter list, and the names of those parameters essentially form a contract (Ian Whitlock's term) with the user. So, it's important that those names and that parameter list correctly and sufficiently describe the scope and use of the macro.

If the **%addMonthFolder()** macro is going to be more generally useful, we need to be able to add directories other than simply *yyyymm* for the date of the report. Really we want a macro that will assure the user that a requested path exists. Let's now call this macro **%assuredPath()**.

Parameters

Parameters can describe details of this contract. It also allows more flexibility so that the macro is more generally useful. Think of them as those underlined blanks in a boilerplate contract.

Our confirmed path needs to start from a known position. It can be something really basic like **C:**, but the user does need to specify in general where that confirmed path starts. We will pass this known position as a parameter; let's call it **root=**. Then we need to specify the part of the path that must be verified and possibly created. Let's pass that in with **branch=**. Whenever possible, our parameters should have default values. Those default values allow the macro to run if the user does not specify the parameters and as such should be reasonable. If this macro specifies a directory for output and the user calls the macro without parameters, this job will run and the user can find the output. A dot means "right here" so let's use that for the default for the root because "right here" always exists. For the branch, let's name the directory with the date and time at which the directory was created. The **BN8601DT19.** format writes datetime values as *yyyymmddThhmmss* and this is perfectly acceptable to Windows as a directory name.

```
%macro assuredPath(
    root=.
    , branch=%sysFunc( putN( %sysFunc( dateTime() ) , b8601dt19. ) )
) ;
```

We need parameters to allow flexibility for both known and suspected variations in the way our macro might be used. Be generous with these parameters; if you give them a default, the user does not have to specify them, but the macro will be more flexible if some of those specifics are accessible to change. For example, suppose I have a macro that runs **Proc freq**. We can specify options in the table statement, such as:

```
Tables height*weight / list missPrint ;
```

Of course, this could be hardcoded in our macro, but our macro would be more generally useful if it included a parameter: **tableOptions= list missPrint**.

For this macro, we do not foresee any other parameters that might be useful. However, we typically add a parameter – **testing=0** – to our macros which usually tells the macro to do things like write messages to the log or keep variables it normally would not keep or do other useful things to help when we are in testing mode. That parameter probably has no function in this macro, but let's add it anyway. Our users

are accustomed to being able to specify it; we might later find a use for it. Having it there as a stub really does no harm and will allow for future transparent implementation.

```
%macro assuredPath(
    root=.
    , branch=%sysFunc( putN( %sysFunc( dateTime() ) , $n8601ba16. ) )
    , testing=0
) ;
```

Our macro uses the **dCreate()** function which has two parameters, the root folder and the folder to be created. The root folder has to exist or else the function will throw an error. Therefore, our macro needs to check to see if that folder really exists and if not, it needs to exit gracefully and give the user the information needed to fix the problem. The following code should be added to address this need.

```
%if not %sysFunc( fileExist( &root ) )
    %then %do ;
        %message(
            type= E R R O R
            , text= Directory &root\ cannot be found.
        )
        %let fatalError = 1 ;
        %return ;
    %end ;
%else %do ;
```

The definition for the macro **%message()** is located in the appendix. If the **&fatalError** macro variable exists where this macro is called, it can be used to stop processing after returning from this macro. It needs to be initialized to **0**.

Our original macro simply added a single folder, but we are now allowing the verification and possible addition of an entire branch. See *Appendix 1* for the entire code.

The **helpRequested** parameter

In general, macros are more flexible and more self-documenting with keyword parameters rather than positional parameters. We have found one notable exception – an exception that works only if it is the sole positional parameter, and we use that extensively.

We like to use a positional parameter called **helpRequested** in our macros. The macro then tests for the length of the value for that parameter, and if it has a length the macro does nothing other than write messages to the SAS log about the use of the macro. If the macro is called with only the keyword parameters, the **helpRequested** parameter passes nothing and thus has no length.

```
%macro assuredPath(
    helpRequested
  , root=.
  , branch=%sysFunc( putN( %sysFunc( dateTime() ) , $n8601ba16. ) )
  , testing=0
) ;
```

Now, suppose our user wants to know how to use this macro. (We will discuss how to inform them of the existence of the macro later.) The user can simply submit **%assuredPath(help)** or **%assuredPath(?)** or **%assuredPath(what gives)** or any value for that positional parameter and the macro will kick into teaching mode.

```
/* If the user requested help, we need to provide syntax help. */
%if %length(&helpRequested) %then %goTo helpMsg ;
```

We put the help messages at the bottom of the macro.

```
%helpMsg:
  %put NOTE: %nrStr(%assuredPath( root= , branch= ) ) ;
  %message(
    type= N O T E
    , text=
      This macro will return a path. If the entire path
      does not exist%str(,) it will create the missing
      folders.
  )
```

We output more messages about the parameters and preconditions, etc. – essentially most of the information in our macro header. See *Appendix 1*.

Compiled Macros

We can save our macros in a compiled macro catalog. We can save the source code there, too. SAS tells of sophisticated methods for compiling the macros into a catalog and saving the source code. However, in a windowing environment, we found it much simpler to simply run the macro which creates a macro catalog in the work library and compiles the macro into that catalog. If we don't already have a permanent compiled macro catalog, we can simply copy this catalog from work to our permanent location. If we already have a permanent macro catalog, we can drag the compiled macro from work.sasmacr to our permanent macro catalog. We can save the source code in the same catalog by creating a new *Source Program* object and copying the macro code into it. To modify the macro, we simply open the source code in the *Source Program* object, make the changes, run it, and copy the newly compiled macro from work.sasmacr to our permanent macro catalog.

When we access our catalog for production use, the library needs to have read-only access. Otherwise only one user can access it at a time. The users will need to specify something like the following in their SAS code.

```
LibName macroCat "\\server\MacroStore\" access=readOnly ;  
Options mStored sasMStore=macroCat mAutoSource ;
```

But we really don't want to ask them to include this into all of their programs. So we should implement an *autoExec* file with this code.

We really save very little run time when we use a compiled macro catalog. That's not the advantage. We do gain a great deal of organization and version control. Instead of having dozens or hundreds of separate files of source code, we can have one macro catalog. That can prove much easier to manage.

The **%help()** Macro

We find it useful to provide a **%help()** macro. That macro simply writes the names of all of the macros in our macro catalog to the SAS log with a very short description. The user can then get more information by calling the macro with a value for the positional **helpRequested** parameter.

```

%macro help() ;
  %put %sysFunc( repeat( = , %sysFunc( getOption(lineSize) )-1 ) ) ;
  %put NOTE: Any of the macros in this compiled macro catalog can;
  %put NOTE- be called passing help to get usage details. E.g.: ;
  %put ;
  %put NOTE-      %nrStr(%anyEq(help)) ;
  %put ;
  %put NOTE- Compiled macros in this catalog: ;
  %put NOTE- %nrStr(%anyEq( varList= , valueList= )) ;
  %put NOTE- %nrStr(%char2num)(
  %put NOTE-      data=
  %put NOTE-      , out=
  %put NOTE-      , charVars=
  %put NOTE-      , inFormat=
  %put NOTE- ) ;
  %put NOTE- %nrStr(%confirmedPath( knownPath= , desiredBranch= )) ;
  %put NOTE- %nrStr(%dedup( data= , out= , key= , dupOut= )) ;
  %put NOTE- %nrStr(%getNObs( data= )) ;
  %put NOTE- %nrStr(%getOrderedVarNames)( ;
  %put NOTE-      data= ;
  %put NOTE-      , quoted= ;
  %put NOTE-      , commaDelimited= ;
  %put NOTE- ) ;
  %put NOTE- %nrStr(%listCount( mList= )) ;
  %put NOTE- %nrStr(%message( text= , type= )) ;
  %put NOTE- %nrStr(%num2char( data= , out= , numVars= , format= )) ;
  %put NOTE- %nrStr(%reportUnexpectedNObs)( ;
  %put NOTE-      data= ;
  %put NOTE-      , expected= ;
  %put NOTE-      , msgLevel= ;
  %put NOTE- ) ;
  %put NOTE- %nrStr(%title( text= )) ;
  %put ;
  %put NOTE- These macros have an undocumented parameter ;
  %put NOTE- (testing=) that will often write processing ;
  %put NOTE- information to the SAS log when called with testing=1.;
  %put NOTE- The default is testing=0. ;
  %put %sysFunc( repeat( = , %sysFunc( getOption(lineSize) )-1 ) ) ;
%mEnd help ;

```

Conclusion

Macro catalogs and autocall libraries are very valuable. However, the standards for a macro that others will use is far higher than the standards for macros that exist in your programs. Make them user friendly with descriptive names for the macros and their parameters. Use default values whenever possible, and provide sufficient parameters for flexibility. A positional help parameter can make your macro more user-friendly. Finally, consider a %help() macro in your autocall library or macro catalog. Follow these tips and you will go far in creating macros that others want to use.

Finally, consider a compiled macro catalog. This can make administration of your macros more manageable.

Contact Information

Your comments and questions are valued and encouraged. Contact the authors at:

Ed Heaton
Data and Analytic Solutions, Inc.
3057 Nutley Street
Fairfax, VA 22031
Voice: 301-520-7414
Email: EHeaton@DASConsultants.com

Sarah Woodruff
Westat
1600 Research Boulevard
Rockville, MD 20850
Voice: (240) 314-7562
Email: SarahWoodruff@Westat.com

The content of this paper is the work of the authors and does not necessarily represent the opinions, recommendations, or practices of Data and Analytic Solutions, Inc. or Westat. SAS[®] and all other SAS Institute Inc. product or service names are registered trademarks or trademarks of SAS Institute Inc. in the USA and other countries. ® indicates USA registration.

Other brand and product names are trademarks of their respective companies.

Appendix 1

```

/*****
MACRO: %assuredPath()

OBJECTIVE:
    This macro will return a path.  If the directory does not exist,
    it will be created.

PREREQUIRMENTS:
    LibName macroCat "\\server\MacroStore\" access=readOnly ;
    Options mStored sasMStore=macroCat mAutoSource ;

PROGRAMMER:
    Ed Heaton, SAS Senior Systems Analyst
    SAS® Certified Advanced Programmer for SAS® 9
    Data & Analytic Solutions, Inc.,
    Voice: (301) 520-7414                Fax: (703) 991-8182
    mailto:eheaton@dasconsultants.com   http://www.dasconsultants.com

EXAMPLES:
    LibName foo "%assuredPath(
        root=%sysGet(TEMP)
        , branch=foo\child\grandchild
    )" ;
    LibName xlsLib "%assuredPath(
        root=\\server\project
        , branch=Data
    )\MyWorkbook.xls" ;
    %local fatalError ;
    Ods pdf file="%assuredPath(
        root=..
        , branch=Reports
    )\MyReport.pdf" ;
    %if &fatalError %then %return ;

STORAGE:
    This source code and its compiled macro are stored in
    \\server\MacroStore\sasmacr.sas7bcat.  A copy of the source code
    is kept in \\ server\MacroStore\assuredPath.sas.

AUDIT TRAIL:

```

20080611 EH Developed macro to return a valid path, even if part of the tree does not already exist. It will create the missing parts.

20081104 EH Added a positional parameter for help.

20081124 EH Replaces some %PUT statements with calls to the %message() macro.

```

*****/
%macro assuredPath(
  helpRequested
  , root=.
  , branch=%sysFunc( putN( %sysFunc( dateTime() ) , $n8601ba16. ) )
  , testing=0
) ;

/* If the user requested help, we need to provide syntax help. */
%if %length(&helpRequested) %then %goTo helpMsg ;
/* If &fatalError exists where this macro is called, it can be used
to stop processing after returning from this macro. */
%let fatalError = 0 ;

%if not %sysFunc( fileExist( &root ) )
  %then %do ;
    %message(
      type= E R R O R
      , text= Directory &root\ cannot be found.
    )
    %let fatalError = 1 ;
    %return ;
  %end ;
%else %do ;
  %local i ; %let i = 1 ;
  %do %while ( %length( %scan( &branch , &i , \ ) ) ) ;
    %if %sysFunc(
      fileExist( &root\%scan( &branch , &i , \ ) )
    ) %then %let root =
      &root\%scan(&branch,&i,\)
    ;
  %else %do ;
    /* If this folder does not exist, create it and
    add the new folder name to the path in the
    &root macro variable. */

```

```

%local newPath ;
%let newPath = %sysFunc( dCreate(
    %scan( &branch , &i , \ )
    , &root
) ) ;
/* Test for success. */
%if %length(&newPath)
%then %do ;
    %message(
        type= W A R N I N G
        , text= &newPath does not exist.
    )
    %let root = &newPath ;
    %message(
        type= N O T E
        , text=&root created.
    )
%end ;
%else %do ;
    %message(
        type= E R R O R
        , text=
            &root\%scan(&branch,&i,\)
            does not exist and could not
            be created.
    )
    %let fatalError = 1 ;
    %return ;
%end ;
%end ; /* %else ... */
%let i = %eval( &i + 1 ) ;
%end ; /* %do %while ... */
%end ;
%unquote(&root)
%return ;
/* ----- */
%helpMsg:
%local look ; %let look = TE ; %let look = NONOTE ;
%put %sysFunc(getOption(lineSize)) ;
%put %sysFunc( repeat(=,%sysFunc(getOption(lineSize))-1) ) ;
%put NOTE: %nrStr(%assuredPath( root= , branch= )) ;

```

```

%message(
    type= N O T E
    , text=
        This macro will return a path. If the entire path
        does not exist%str(,) it will create the missing
        folders.
    )
%message(
    type= N O T E
    , text=
        This macro will set the %nrStr(&fatalError) macro
        variable to 1 if there is a problem getting the path
        or 0 if there is no problem. If that macro variable
        exists where this macro is called%str(,) it can be
        used to stop processing after returning from this
        macro. If it does not exist there%str(,)
        %nrStr(&fatalError) will be local to this macro and
        will have no effect elsewhere.
    )
%message(
    type= N O T E
    , text=This macro is valid anywhere.
    )
%message(
    type= N O T E
    , text=
        The root= parameter specifies a folder that is known
        to exist. The default is a dot.
    )
%message(
    type= N O T E
    , text=
        The branch= parameter specifies a branch from the
        folder that might exist. If it does not%str(,)
        this macro will create it.
    )
%put NOTE: Examples: ;
%put NOTE-      LibName foo %str("%")%nrStr(%assuredPath)( ;
%put NOTE-      root=%nrStr(%sysGet)(TEMP) ;
%put NOTE-      , branch=foo\child\grandchild ;

```

above

```

%put NOTE-      )%str("%" ;) ;
%put NOTE-      LibName xlsLib %str("%")%nrStr(%assuredPath)( ;
%put NOTE-      root=\\server\project ;
%put NOTE-      , branch=Data ;
%put NOTE-      )\MyWorkbook.xls%str("%" ;) ;
%put NOTE-      %nrStr(%local) fatalError %str(;) ;
%put NOTE-      Ods pdf file=%str("%")%nrStr(%assuredPath)( ;
%put NOTE-      root=.. ;
%put NOTE-      , branch=Reports ;
%put NOTE-      )\MyReport.pdf%str("%" ;) ;
%put NOTE-      %nrStr(%if &fatalError %then %return ;) ;
%put %sysFunc( repeat( = , %sysFunc(getOption(lineSize))-1 ) )
;
/***** End of Macro *****/

```

Appendix 2

```

/*****
MACRO: %message()

OBJECTIVE:
    This macro will write a NOTE, WARNING, or ERROR message to the SAS
    log. It will partition the message so that it fits on a line and
    is properly indented.

PREREQUIRMENTS:
    LibName macroCat "\\server\MacroStore\" access=readOnly ;
    Options mStored sasMStore=macroCat mAutoSource ;

PROGRAMMER:
    Ed Heaton, SAS Senior Systems Analyst
    SAS® Certified Advanced Programmer for SAS® 9
    Data & Analytic Solutions, Inc.,
    Voice: (301) 520-7414          Fax: (703) 991-8182
    mail:ehaton@dasconsultants.com http://www.dasconsultants.com

STORAGE:
    This source code and its compiled macro are stored in
    \\server\MacroStore\sasmacr.sas7bcate. A copy of the source code
    is kept in \\ server\MacroStore\message.sas.

```

AUDIT TRAIL:

20080910 EH Developed macro to aid in the formatting of messages to the SAS log.

20081107 EH Added a positional parameter for help.

```
*****/
%macro message( helpRequested , text= , type=N O T E , testing=0 ) ;

    %local look ; %let look = TE ; %let look = NO&look ;
    %local hmmm ; %let hmmm = RNING ; %let hmmm = WA&hmmm ;
    %local ohNo ; %let ohNo = ROR ; %let ohNo = ER&ohNo ;

/* If the user requested help, we need to provide syntax help. */
%if %length(&helpRequested) %then %goTo helpMsg ;

/* If the &fatalError macro variable exists where this macro is
called, it can be used to stop processing after returning from
this macro. If it doesn't exist there, that's okay; &fatalError
will be local to this macro and will have no effect. */
%let fatalError = 0 ;

/* If the user specified the text of the message on multiple indented
lines, there will be excessive blanks that need to be removed. */
%let text = %cmpres(&text) ;
%if &testing %then %put &look: %nrStr(&text)=&text ;

/* The message type needs to be N O T E, W A R N I N G, or E R R O R
-- all upper case and without the embedded spaces. */
%let type = %upCase( %sysFunc( compress(&type) ) ) ;
%if &testing %then %put &look: %nrStr(&type)=&type ;
%if not (
    ( &type eq &look )
    or ( &type eq &hmmm )
    or ( &type eq &ohNo )
) %then %do ;
    %put &ohNo: type= must pass &look, &hmmm, or &ohNo.. ;
    %return ;
%end ;

/* Determine the length of the part of the message that will fit on a
line. It varies with the lineSize= system option and the message
type. */
```

```

%local partitionLength ;
%let partitionLength = %eval(
    %sysFunc( getOption(lineSize) )
    - %length(&type)
    - 2
) ;
%if &testing
    %then %put &look: %nrStr(&partitionLength)=&partitionLength ;

/* If the message will fit on a line, then write it.  Otherwise,
partition it and write each partition. */
%if ( %length(&text) le &partitionLength )
    %then %put &type: &text ;
%else %do ;
    /* Partition the message at spaces or -- if we have a path --
at a slash (hack). */
    %local partitionBreak ;
    %let partitionBreak = %sysFunc( findC(
        %superQ(text)
        , %str( \ )
        , -&partitionLength
    ) ) ;
    %put &type: %substr( %superQ(text),1,&partitionBreak-1 ) ;
    /* Now remove the part of the message that we wrote from the
message, leaving only the part that remains. */
    %let text = %substr( %superQ(text) , &partitionBreak ) ;
    /* Continue writing segments of the message until what is
left will fit on a line. */
    %do %while ( %length(%superQ(text)) gt &partitionLength ) ;
        %let partitionBreak = %sysFunc( findC(
            %superQ(text)
            , %str( \ )
            , -&partitionLength
        ) ) ;
        %put &type-%substr(%superQ(text),1,&partitionBreak-1);
        %let text = %substr( %superQ(text) , &partitionBreak);
    %end ;
    /* Write the last of the message. */
    %put &type- &text ;
%end ;

```

```

%return ;

/* ----- */
%helpMsg:
  %local look ; %let look = TE ; %let look = NO&look ;
  %put %sysFunc(repeat( = ,%sysFunc(getOption(lineSize)) - 1)) ;
  %put &look: %nrStr(%message)( text= , type= ) ;
  %put ;
  %put &look- This macro will write a NOTE, WARNING, or ERROR;
  %put &look- message to the SAS log. It will partition the;
  %put &look- message so that it fits on a line;
  %put &look- and is properly indented.;
  %put ;
  %put &look- Valid anywhere a %nrStr(%PUT) statement is valid.;
  %put ;
  %put &look- Parameter: ;
  %put &look-text= specifies the text of the message. This can ;
  %put &look- lines and can be indented because multiple ;
  %put &look- white-space will be removed. ;
  %put &look- type= specifies the type of message (NOTE, ;
  %put &look- WARNING or ERROR).
  %put &look- Any embedded blanks will be removed. ;
  %put ;
  %put &look- Examples: ;
  %put &look- %nrStr(%message)( ;
  %put &look- text=This macro seems to work just fine. ;
  %put &look- , type=N O T E ;
  %put &look- ) ;
  %put ;
  %put &look- %nrStr(%message)( ;
  %put &look- text= ;
  %put &look- This message is longer than will fit
  %put &look- on a line. This macro will remove ;
  %put &look- all repetitive white space -- ;
  %put &look- including spaces%str(,) tabs%str(,) ;
  %put &look- and carriage returns%str(,) and ;
  %put &look- then repartition the output according;
  %put &look- to the specifications of the ;
  %put &look- LINESIZE= system option. ;
  %put &look- , type=W A R N I N G ;
  %put &look- ) ;

```

```
%put ;
%put &look-      %nrStr(%message)( ;
%put &look-      text= ;
%put &look-      This message brought to you by ;
%put &look-      %nrStr(%sysFunc)( dequote( ;
%put &look-      %nrStr(%substr(&sysProcessName,8)) ;
%put &look-      ) ). ;
%put &look-      , type=N O T E ;
%put &look-      ) ;
%put %sysFunc( repeat(=,%sysFunc(getOption(lineSize))-1) ) ;
%put ;
%*mEnd message ;
/***** End of Macro *****/
```