Paper BB-01

# %whatChanged: A Tool for the Well-Behaved Macro

Frank DiIorio, CodeCrafters, Inc., Philadelphia PA

The power and usefulness of macros is undeniable. Also above dispute is the ability of a poorly constructed macro to introduce chaos in programs that use it. Many of these undesirable side effects can be prevented by following some simple design and coding practices.

One of these practices, and the focus of this paper, is that the macro should produce only what is described in its documentation. If, for example, it is supposed to create a dataset and a macro variable, then these items should be the only changes to the SAS® environment once the macro terminates. There are no built-in SAS tools that compare "snapshots" of resources and settings. Not surprisingly, `%whatChanged`, the macro described in this paper, addresses this need.

This paper very briefly discusses some principles of good macro design. It then describes the design, coding, and use of `%whatChanged`. While discussion of the macro is useful in and of itself, the other goal of the paper is to give the reader some insight and motivation to develop similar tools.

The paper is appropriate for anyone charged with developing, debugging or enhancing macros. A basic knowledge of the macro language is assumed.

## Introduction

Consider the following scenario. You want to identify variables in a dataset that have no non-missing values. You review the tools in your company's macro library and find `%missVars`. Its header documentation describes input and output-related parameters. It also identifies the lone item that it creates: a SAS dataset specified by the macro's `OUT` parameter. After reading the documentation, you run the macro and print its output dataset.

```
%missVars(data=clin.ae, out=ae_miss, keep=usubjid)
proc print data=ae_miss;
     id usubjid;
     title "Variables in CLIN.AE that had all missing values";
run;
```

As expected, `%missVars` created dataset `AE_MISS`. You later discover that `%missVars` also created several temporary datasets and created several global macro variables. In this program these artifacts were not problematic (they didn't throw off dataset counts or overwrite like-named variables). It is easy to see, though, that they could easily have negative consequences. If, however, `%missVars` was a utility that was deeply nested in an application, the potential for disruption and subsequent tedious debugging would increase dramatically.

Even this simple scenario suggests that any macro should adhere to a set of design, programming, and validation best practices. This is vital to ensure that the tool is solid and reliable by *design* and not simply by *chance*. The best practice that we focus on in this paper is that validation of a widely used, production-quality macro should test for the presence of what was documented and for the absence of unintended outputs.

## Keys to Good Design

While macro design is not the focus of this paper, it *is* both an important topic and relevant to the creation of `%whatChanged`. The following items are taken from *Building the Better Macro: Best Practices for the Design of Reliable, Effective Tools* (see "Contact," below, for location).

- **Know when a macro *is* and *is not* necessary.** Code written as a macro can sometimes be replaced by `CALL EXECUTE`, open-code use of `%SYSFUNC`, SQL-generated macro variables, BY-group processing or similar, more transparent and easily maintained constructs.
- **Use keyword parameters and values consistently.** Parameter naming and values should be consistent within a system of macros. Parameter values should be standardized (e.g., upper-cased) early in the program to facilitate evaluation in `%IF` and similar statements.
- **Use consistent program structure.** Predictable program structure brings new (non-developer) programmers up to speed more quickly than idiosyncratic construction. This consistency is particularly important during debugging, when the speed of problem isolation is critical.

- **Know when to redesign.** If a macro becomes extremely large, filled with complex coding constructs and numerous indirect variable references (the dread `&&var`, `&&&var`, etc.) consider taking a different, possibly simpler approach.
- **Emphasize user and between-program communication.** Control of messages relevant for both the end user and debugging programmer is essential. This can take various forms, among these: routine and error/warning text written to the SAS Log, status variables and error logs.
- **Take control of macro variable scope.** Use of `%LOCAL` and `%GLOBAL` statements is simple, and removes any ambiguity about which values are known to which macros. This is especially critical when macros use specialized, single-purpose utilities.
- **Implement diagnostic and debugging code.** At any point in the life cycle, but especially during development, it is helpful to have a means to control the amount of debugging and diagnostic output produced by the macro.
- **Use built-in macro tools.** Be aware of what built-in tools SAS has to offer, and be prepared to build the tools you need.

Finally, we have the design features most relevant to this paper:

- **Clearly document the macro.** Headers and other comments are the programmer's and user's key to understanding program logic. The header block should: describe the macro's purpose; identify parameters; present examples; note naming conventions; present examples of use; describe complex program logic; and document revisions. Most relevant to this paper, the header should identify macro output: macro variables, datasets, text files in varying, and the like.
- **Produce only what was promised.** Amplifying the previous point, the header contains the complete enumeration of all items created by the macro. *These items should be the only legacy of the macro's execution. Creating anything else gives the perception of sloppy coding, and raises the possibility of contamination of the program that called the macro.*

## Why the Macro Is Needed

There are numerous macro-related functions, statements, and autocall macros available in the out-of-the-box version of SAS software. In complex applications, however, these features are usually just the building blocks for specialized, home-grown diagnostic tools. The macro developer must devote some time to creating utilities that help ensure that the macro is performing correctly. With "only produce what was promised" in mind, let's turn our attention to the development of `%whatChanged`.

Consider this excerpt from the header of macro `%AEsummary`:

```
Output:   Datasets AE_SUMM and AE_DETAIL
          Global macro variables NPAT, NPAT_NOAE, NPAT_AE
```

The macro programmer is promising – *and the user is expecting* – that when the macro terminates, the SAS session will be changed by the addition of two datasets and three global macro variables. It is relatively easy for the programmer to produce what was expected. What is also needed is a programmatic approach to verifying results, a way to compare before and after "snapshots" of the SAS session.

In our example, the developer needs to ask several questions. What datasets and global macro variables were defined before and after the macro was called? If the dataset existed prior to the macro call, was there a change in the number of observations or variables? Most importantly, did these snapshots of the SAS session differ *only by the datasets and macro variables that were expected?*

Let's turn our attention to the macro's design.

## What Needs to be Done? Logic and Structure

### Multiple Calls

Our before-and-after comparison requirement makes the macro's coding and execution a bit different from most other utilities. Rather than a single invocation, `%whatChanged` needs to be invoked twice:

```
%whatChanged   Take "before" snapshot of SAS session
%macroToTest   Run the macro being tested
%whatChanged   Take "after" snapshot of SAS session, then compare to "before" snapshot
```

The first call to `%whatChanged` takes the "before" picture of the SAS session's datasets and global macro variables. The macro being tested executes, then `%whatChanged` is called again. This time it takes the "after" picture and compares datasets and global macro variables. Let's take a high-level look at what the macro processes and what is required by each call.

### First Call

Calling `%whatChanged` before the macro being tested places the following requirements on the developer:

- The macro should execute in open code. Test for this, and write a message and terminate if the user invoked it incorrectly.
- Evaluate parameter `ACTION`. If upper-cased values is not `START` or `END`, write a message to the Log and terminate.
- If `ACTION=START`, allocate `LIBNAME _DELTA_`.
- Create `_DELTA_.DATASETS_PRE` and `_DELTA_.MACROVARS_PRE` from `DICTIONARY.TABLES` and `DICTIONARY.MACROS`, respectively.

### Second Call

Calling `%whatChanged` after the macro being tested is a bit more involved:

- Repeat the open code test and `ACTION` evaluation described in “First Call,” above.
- If `ACTION=END` and `LIBNAME _DELTA_` exists, then create `_DELTA_.DATASETS_POST` and `_DELTA_.MACROVARS_POST` from `DICTIONARY.TABLES` and `DICTIONARY.MACROS`, respectively.
- In separate DATA steps, one for macros, another for datasets, merge the `xxx_PRE` and `xxx_POST` datasets.
- Report differences between the datasets. The report identifies new, deleted, or modified datasets and macro variables. This requires a combination of using `IN` dataset options and, for observations in both pre and post, variable-by-variable comparisons (comparing, for example, the Dictionary Table abstract’s `NOBS` variable in the pre and post abstracts to see of the number of observations has changed).
- Clear `LIBNAME _DELTA_`.

This is a quick overview of the macro’s structure. The next section describes the “film” for the pre and post session “snapshots.”

## What Tools Are Required?

### A Quick Review of Dictionary Tables

**The Big Picture.** With the simplified design of the previous section in mind, attention naturally turns to what, exactly, is being used for the comparisons. How do you capture dataset names and characteristics, macro variables, and other items of interest. You might, for example, want to search for TITLE, FOOTNOTE, and LIBNAME differences. This, and much more information about the SAS session, is automatically stored and maintained in the SAS Dictionary Tables. Here are some key features of the Tables**:**

- The Tables and a set of views in the `SASHELP` library are automatically created and populated whenever you start an interactive or batch SAS program. No setting of system options or initialization is required.
- They are protected and read-only. You can’t deallocate them or change their structure.
- The contents of the Tables are influenced by what occurs in the SAS session. If you create an indexed dataset with 20 variables, for example, one or more rows are added to the `INDEXES` table, a row is added to the `TABLES` table, and 20 rows are added to the `COLUMNS` table.
- The Tables are accessible only from PROC SQL, using the `LIBNAME DICTIONARY` (yes, that’s a 10-character `LIBNAME`). To access the Tables’ contents outside SQL, use a set of Views that are allocated in `SASHELP`. With few exceptions (none of which are relevant for the purposes of this paper), there is a one-to-one correspondence between Tables and Views.
- Discovering what, exactly, is in the Tables and Views is not a straightforward process, but *is* simplified somewhat by SAS online help, and SAS conference papers. Also see the reference to the CodeCrafters, Inc. website in “Contacts,” below.
- The Tables and Views are updated whenever a part of the SAS environment changes: if a dataset is created, a row is added to the `TABLES` table, a row is added to the `COLUMNS` table for each variable in the datasets, and if the dataset is indexed, the `INDEXES` table is updated.

**The Focused Picture.** Since `%whatChanged` tracks changes to macro variables and datasets, we will limit the discussion to Tables and Views that contain information about these items.

The "Output" section of our test case macro said it would produce three global macro variables. In order to create lists of what was present before and after a call to the macro, we need to capture rows from the `MACROS` table. Its contents are described in **Figure 1**, below.

**Figure 1: DICTIONARY.MACROS, SASHELP.VMACRO**

| Variable | Type/Length | Description |
|---|---|---|
| scope | $9 | Macro scope [GLOBAL\|AUTOMATIC\|macro name (if local)] |
| name | $32 | Macro variable name (upper-cased) |
| offset | num | Offset into macro variable value [0, 200, …]. Note that the beginning of a value spanning observations may not always be stored with OFFSET=0. |
| value | $200 | Macro variable value (case and spacing are preserved) |

Note the following about `MACROS`:

- Variable `SCOPE` can be used to select Global macro variables (value "`GLOBAL`")
- The name of the variable is always upper-cased.
- Since the value of each macro variable is broken into 200-character pieces, variables whose values are longer than 200 characters will be stored in more than one row. This is where the `OFFSET` variable comes into play. Selecting `OFFSET` values of 0 ensures that the list of macro variables will be unique.

Next, let's consider the dataset that was other item in the "Output" section of our test case macro. To create lists of datasets present before and after a call to the macro, we need to capture rows from the `TABLES` table (**Figure 2**, below). This is a considerably richer table than `MACROS`. As is the case with any of the Tables, it's possible to go for years without using a particular column. What is important here, as you are becoming familiar with this resource, is simply to know that these items exist. That way, if there is occasion for their use you will be able to quickly identify the table and column you need and put this session metadata to use.

**Figure 2: DICTIONARY.TABLES, SASHELP.VTABLE**

| Variable | Type/ length | Description | Value for non-native engine |
|---|---|---|---|
| libname | $8 | Library name (upper case) | |
| memname | $32 | Member name (*may* be case-sensitive and include embedded blanks for non-native engines). | |
| memtype | $8 | Member type [DATA\|VIEW] | |
| memlabel | $256 | Dataset label | blank |
| typemem | $8 | Dataset type (upper case) [blank\|MSGFILE\|ATTLIST\|…] | |
| crdate | num | Date created. Uses DATETIME format and informat. | . |
| modate | num | Date modified. Uses DATETIME format and informat. | . |
| nobs | num | Number of observations | . |
| obslen | num | Observation length | 0 |
| nvar | num | Number of variables | |
| protect | $3 | Password protection. Position 1: -\|R, Position 2: -\|W, Position 3: -\|A. | --- |
| compress | $8 | Compression routine [NO\|CHAR\|BINARY] | NO |
| encrypt | $8 | Encryption used? YES\|NO | NO |
| filesize | num | File size [>= 0] | 0 |
| npage | num | Number of allocated pages (>= 0) | 0 |
| pcompress | num | Percent compression (integer. Can be negative) | . |
| reuse | $3 | Reuse space? [no\|yes] | no |
| bufsize | num | Buffer size | 0 |
| delobs | num | Number of deleted observations (>= 0) | 0 |
| indxtype | $9 | Type of indexes [blank\|SIMPLE\|COMPOSITE\|BOTH] | |
| datarep | $32 | Data representation | |
| reqvector | $24 | Requirements vector. Uses $HEX format and informat. | |
| nlobs | num | Number of logical observations [. if view, else >= 0] | . |
| maxvar | num | Length of longest variable name | 0 |
| maxlabel | num | Length of longest variable label | 0 |
| maxgen | num | Maximum number of generations | 0 |
| gen | num | Generation number [., 0, 1, 2, …] | . |
| attr | $3 | Dataset attributes | |
| sortname | $8 | Name of collating sequence | |
| sorttype | $4 | Sorting type S:sort verified, SR:sort used NODUPREC, SK: sort used NODUPKEY | |

| | | | |
|---|---|---|---|
| sortchar | $8 | Character set used in sort [ANSI ASCII] | |
| datarepname | $170 | Data representation name | |
| encoding | $256 | Data encoding (blank if view) | |
| dbms_memtype | $8 | DBMS member type [VIEW\|TABLE] Blank if not using DBMS engine such as MDB, XLS. | |

For use with `%whatChanged`, note the following about `TABLES`:

- Variables `LIBNAME`, `MEMNAME`, and `MEMTYPE` uniquely identify a table. These are, in effect, the BY variables that will be used when we compare our pre- and post-snapshot files.
- There are many attributes that `%whatChanged` could compare. Some of the more common ones are modification date (`MODATE`), number of variables (`NVAR`) and number of observations (`NOBS`).

### Other Tools

Dictionary Tables and Views are essential tools for `%whatChanged` and many other utility macros. Some familiarity with other, less arcane and better-publicized features of the macro language is also helpful. The following table identifies macro language features that are relevant for adhering to the “create only what was promised” theme being developed in this paper. The table is brief, emphasizing how the macro feature can be applied rather than delving into syntax specifics.

| **Feature** | **Description** | **Example** |
|---|---|---|
| `%local`<br>`%global` | Control the scope of a macro variable | *Create output macro variables*<br>`%global freqs freqsN;`<br>`%let freqsN = -1;`<br>*Declare `%DO` loop index and other temporary variables:*<br>`%local i j token;` |
| `%symdel` | Delete a macro variable, optionally specifying the variable’s scope. | *Delete a global macro variable created during execution:*<br>`%symdel TMPn TMPlist;` |
| `%goto`<br>`%`*`label`* | Branch to a labeled location in a program. | *Branch to common termination location if error arises:*<br>`%if %sysfunc(exist(summary)) = 0 %then %do;`<br>`    %put ERROR: can’t locate SUMMARY;`<br>`    %goto bottom;`<br>*... other statements …*<br>`%bottom:` *end of macro cleanup, messages* |
| `proc delete`<br>`proc datasets` | Delete one or more datasets. | `proc delete data=SUMM1 TRANSPOSE;`<br>`run;` |
| `%sysfunc`<br>`%qSysfunc` | Use system functions in `%IF-%THEN-%ELSE` and assignment statements. | *Save, reset, then restore an option:*<br>`%local obs;`<br>`%let obs = %sysfunc(getoption(obs));`<br>`options obs=max;`<br>*... other statements …*<br>*Restore OBS value in common termination section*<br>`%bottom: options &obs.;` |

## Coding the Macro

A look at the development process for tools like `%whatChanged` can be as interesting at the tool itself. Rather than present the final (or, at least, Version 1) program, instead it is presented in the same steps that the macro developer might follow. This may give the reader some insight into what tasks need to be addressed first. Program source for all versions of the macro is available at the CodeCrafters, Inc. web site (see “Contacts,” below, for location).

### Version 0.1: Bare Bones, Assume the Best

The simplest description of the macro is that it builds and compares lists of macro variables and datasets. The first version is limited to this functionality. We assume that parameter `ACTION` had a valid value and that the temporary library will be successfully allocated. Comments about the coding are in shaded text.

```
%macro whatChanged(action=);
       %let action = %upcase(&action.);
```

```
%if &action. = START %then %do;
    Create a separate directory to use for storing datasets needed by the macro.
    %let path = %sysfunc(pathname(work))\_&sysIndex._;
    We set the XWAIT option without considering its value prior to the macro.
    options noxwait;
    The MD command is Windows/DOS-specific.
    x "md ""&path.""";
    libname _delta_ "&path.";
    %end;

Create suffix for non-merge/BY variable names.
%if &action. = START %then %let suffix = pre;
    %else                  %let suffix = post;

This section is run regardless of ACTION value.
proc sql noprint;
     Simplified coding here for sake of brevity. It ignores the possibility that
     a macro variable's change may occur after position 200, in an observation
     with OFFSET > 0.
     create table _delta_.macVars&suffix. as
     select name, value as value&suffix.
     from dictionary.macros
     where scope="GLOBAL" & offset=0
     order by name
     ;
     create table _delta_.dataSets&suffix. as
     Select items to compare that are important to you. Here, we chose # obs,
     # variables, and modification date.
     select libname, memname, memType,
            nobs as nobs&suffix., nVar as nvar&suffix.,
            modate as modate&suffix.
     from dictionary.tables
     Exclude the macro's temporary directory (_DELTA_) and SASHELP.
     where libname notin ("SASHELP", "_DELTA_")
     order by libname, memname, memType
     ;
quit;

%if &action. = END %then %do;

    Merge the PRE/POST macro variable datasets and report differences.
    data _null_;
         merge _delta_.macvarsPre(in=_pre)
               _delta_.macvarsPost(in=_post)
               ;
         by name;
         length comment $20;
         if _pre & _post then do;
            if valuePre ^= valuePost then do;
               comment = 'Value changed';
               link print;
               end;
            end;
         if _pre & ^_post then do;
            comment = "Deleted";
            link print;
            end;
         if ^_pre & _post then do;
            comment = "Added";
            link print;
            end;
         return;

         print: put name comment;
                return;
    run;

    Merge the PRE/POST dataset extracts and report differences.
    data _null_;
```

```
                merge _delta_.datasetsPre(in=_pre)
                      _delta_.datasetsPost(in=_post)
                      ;
                by libname memname memType;
                length comment $40;
                if _pre & _post then do;
                   if nVarPre ^= nVarPost then do;
                      comment = '# of vars changed';
                      link print;
                      end;
                   if nObsPre ^= nObsPost then do;
                      comment = '# of obs changed';
                      link print;
                      end;
                   if modatePre ^= modatePost then do;
                      comment = 'Mod date changed';
                      link print;
                      end;
                   end;
                if _pre & ^_post then do;
                   comment = "Deleted";
                   link print;
                   end;
                if ^_pre & _post then do;
                   comment = "Added";
                   link print;
                   end;
                return;

                print: put libname memname memType comment;
                       return;
           run;

           Clean up after ourselves: clear the LIBNAME.
           libname _delta_ clear;
           %end;
%mend whatChanged;
```

### Version 0.2: Trap Errors, Improve Output, Assume the Worst.

A macro that doesn't check parameter values, leaves variable scope to chance, and ignores other good design principles brings to mind is, as Samuel Johnson said, "the triumph of hope over experience." In this version of the macro, we make minor changes to functionality and add a significant amount of error trapping and messages. Again, comments about the code are in shaded text.

```
%macro whatChanged(action=);

       If not called in open code, write a message and terminate.
       %if &sysprocname. NE %then %do;
           %put Must run in open code.  Execution terminating.;
           %goto lastStmt;
           %end;

       Provide some user feedback.
       %put Begin WHATCHANGED. ACTION=&action.;

       %let action = %upcase(&action.);

       Ensure variables are local to the macro and thus will not overwrite like-named
       and un-scoped variables in the calling environment.
       %local bad i dataset;
       Initialization error trapping.
       Test for valid values of ACTION.
       %if %sysfunc(indexW(START END, &action.)) = 0 %then %do;
           %let bad = t;  %put ACTION [&action.] must be START or END;
           %end;
       Call sequence error: ACTION is START but _DELTA_ already allocated.
       %if &action. = START & %sysfunc(libref(_DELTA_)) = 0 %then %do;
           %let bad = t;  %put ACTION=START, but temporary library was already allocated;
           %end;
```

**Call sequence error: ACTION is END but _DELTA_ was not allocated.**

```
%if &action. = END & %sysfunc(libref(_DELTA_)) ^= 0 %then %do;
    %let bad = t;  %put ACTION=END, but temporary library was not allocated;
    %end;
```

**If BAD was set to t by any test, write message and terminate.**

```
%if &bad. = t %then %do;
    %put Execution terminating for reason(s) noted above.;
    %goto bottom;
    %end;

%if &action. = START %then %do;
    %let path = %sysfunc(pathname(work))\_&sysIndex._;
```

**Rather than overwrite XWAIT setting, capture it, set to NOXWAIT, then restore the original value.**

```
    %local xw;
    %let xw = %sysfunc(getoption(xwait));
    options noxwait;
    x "md ""&path.""";
    libname _delta_ "&path.";
    options &xw.;
```

**Don't assume that the allocation was successful. If not, write message and terminate.**

```
    %if %sysfunc(libref(_DELTA_)) ^= 0 %then %do;
        %put Could not create temporary library. Execution terminating.;
        %goto bottom;
        %end;
    %end;

%local suffix;
%if &action. = START %then %let suffix = pre;
    %else                  %let suffix = post;

proc sql noprint;
     create table _delta_.macVars&suffix. as
     select name, value as value&suffix.
     from dictionary.macros
     where scope="GLOBAL" & offset=0
     order by name
     ;
     create table _delta_.dataSets&suffix. as
     select libname, memname, memType,
            nobs as nobs&suffix., nVar as nvar&suffix.,
            modate as modate&suffix.
     from dictionary.tables
     where libname notin ("SASHELP", "_DELTA_")
     order by libname, memname, memType
     ;
quit;

%if &action. = END %then %do;
```

**See if we have all the datasets needed for the reports. If not, write a message and terminate.**

```
    %do i = 1 %to 4;
        %let dataset = %scan(MACVARSPRE MACVARSPOST
                             DATASETSPRE DATASETSPOST, &i.);
        %if %sysfunc(exist(_delta_.&dataset.)) = 0 %then %do;
           %let bad = t;  %put Could not locate dataset _DELTA_.&dataset.;
           %end;
    %end;
    %if &bad. = t %then %do;
        %put Execution terminating for reason(s) noted above.;
        %goto bottom;
        %end;

    data _null_;
```

**Make the Log messages a bit more informative.  Write a header line, and if there are no discrepancies, say so.**

```
         if _n_ = 1 then put "Compare Macro Variables" / ;
         if eof & _changed = 0 then put "*** No changes detected ***";
         merge _delta_.macvarsPre(in=_pre)
               _delta_.macvarsPost(in=_post)
```

```
           end=eof;
     by name;
     length comment $20;
     if _pre & _post then do;
        if valuePre ^= valuePost then do;
           comment = 'Value changed';
           link print;
           end;
        end;
     if _pre & ^_post then do;
        comment = "Deleted";
        link print;
        end;
     if ^_pre & _post then do;
        comment = "Added";
        link print;
        end;
     return;
```

**_changed is flag used by end of file processing.**

```
     print: _changed + 1;
            put name comment;
            return;
run;

data _null_;
```

**Make the Log messages a bit more informative. Write a header line, and if there are no discrepancies, say so.**

```
     if _n_ = 1 then put "Compare Datasets" / ;
     if eof & _changed = 0 then put "*** No changes detected ***";
     merge _delta_.datasetsPre(in=_pre)
           _delta_.datasetsPost(in=_post)
           end=eof;
     by libname memname memType;
     length comment $100;
     if _pre & _post then do;
```

**Improve on previous version's messages. Rather than report "Value changed", display the actual pre/post values.**

```
        if nVarPre ^= nVarPost then do;
           comment = catX(' ', "# vars changed from",
                          put(nVarPre,5.), "to",
                          put(nVarPost, 5.)) ;
           link print;
           end;
        if nObsPre ^= nObsPost then do;
           comment = catX(' ', "# observations changed from",
                          put(nObsPre,5.), "to",
                          put(nObsPost, 5.)) ;
           link print;
           end;
        if modatePre ^= modatePost then do;
           comment = catX(' ', "Mod datechanged from",
                          put(modatePre,dateTime16.), "to",
                          put(modatePost, dateTime16.)) ;
           link print;
           end;
        end;
     if _pre & ^_post then do;
        comment = "Deleted";
        link print;
        end;
     if ^_pre & _post then do;
        comment = "Added";
        link print;
        end;
     return;

     print: _changed + 1;
            put libname memname memType comment;
            return;
run;
```

```
           libname _delta_ clear;
           %end;

       %bottom is the termination point for both successful and problematic execution (note
       the use of %goto bottom when error condition was raised.
       All termination did here was provide some user feedback. In more complex applications,
       this section could include deletion of temporary datasets, resetting system options,
       clearing filenames, deleting macro variables, etc.
       %bottom: %put End WHATCHANGED;

%lastStmt is branched to if not run in open code.
%lastStmt: %mend whatChanged;
```

### Version 1.0: It’s Not Ready Until You Have Documentation.

The previous version, while robust and functional, is conspicuously devoid of documentation. Since we placed so much emphasis throughout the paper on the importance of documenting expected output, it is reasonable to say that until `%whatChanged` itself has documentation, it isn’t complete. The complete program is shown below. Note that while no run-time functionality has been added, the program does benefit from being well-documented. This facilitates tasks such as debugging and making enhancements.

```
/*
       Name:  whatChanged

Description:  Identify changes to macro variables and datasets that
              occured between invocations of the macro

       Type:  Support

     Inputs:  Parameter (not case-sensitive)
              ACTION   START (collect "before" information) or
                       END (collect "after" information).
                       REQUIRED. No default.

    Outputs:  Messages to location identified as SAS Log

  Execution:  Execute in open code

      Notes:  Fail if any of the following are true:
              [1] ACTION is not START or END
              [2] ACTION=START but temporary library is already
                  allocated.
              [3] ACTION=END but temporary library is not
                  allocated.
              [4] Could not allocate temporary library
              [5] ACTION=END but one or more datasets required
                  for report writing could not be located.

    History:  Programmer       Date        Brief Description
              Frank DiIorio    2010-03-02  Initial program
*/
%macro whatChanged(action=);

       /*** If not running in open code write message and terminate ***/
       %if &sysprocname. NE %then %do;
           %put Must run in open code.  Execution terminating.;
           %goto lastStmt;
           %end;

       %put Begin WHATCHANGED. ACTION=&action.;
       %local suffix bad i dataset xw;

       %let action = %upcase(&action.);

       /*** Error trapping ***/
       %if %sysfunc(indexW(START END, &action.)) = 0 %then %do;
           %let bad = t;  %put ACTION [&action.] must be START or END;
           %end;
       %if &action. = START & %sysfunc(libref(_DELTA_)) = 0 %then %do;
           %let bad = t;  %put ACTION=START, but temporary library was already allocated;
           %end;
```

```
%if &action. = END & %sysfunc(libref(_DELTA_)) ^= 0 %then %do;
    %let bad = t;  %put ACTION=END, but temporary library was not allocated;
    %end;
%if &bad. = t %then %do;
    %put Execution terminating for reason(s) noted above.;
    %goto bottom;
    %end;

/*** START-specific activity ***/
%if &action. = START %then %do;

    /* Put temporary library in a location where we know user
       will have write access */
    %let path = %sysfunc(pathname(work))\_&sysIndex._;

    /* Capture current value of XWAIT */
    %let xw = %sysfunc(getoption(xwait));
    options noxwait;
    x "md ""&path.""";
    options &xw.;     /* Set to XWAIT to original value */

    libname _delta_ "&path.";
    %if %sysfunc(libref(_DELTA_)) ^= 0 %then %do;
        %put Could not create temporary library. Execution terminating.;
        %goto bottom;
        %end;
    %end;


/*** This section done regardless of ACTION value ***/
%if &action. = START %then %let suffix = pre;
    %else                  %let suffix = post;

proc sql noprint;
     create table _delta_.macVars&suffix. as
     select name, value as value&suffix.
     from dictionary.macros
     where scope="GLOBAL" & offset=0
     order by name
     ;
     create table _delta_.dataSets&suffix. as
     select libname, memname, memType,
            nobs as nobs&suffix., nVar as nvar&suffix.,
            modate as modate&suffix.
     from dictionary.tables
     where libname notin ("SASHELP", "_DELTA_")
     order by libname, memname, memType
     ;
quit;

/*** END-specific activity ***/
%if &action. = END %then %do;

    /* First, make sure we have all required datasets */
    %do i = 1 %to 4;
        %let dataset = %scan(MACVARSPRE MACVARSPOST
                             DATASETSPRE DATASETSPOST, &i.);
        %if %sysfunc(exist(_delta_.&dataset.)) = 0 %then %do;
           %let bad = t;
           %put Could not locate dataset _DELTA_.&dataset.;
           %end;
    %end;
    %if &bad. = t %then %do;
        %put Execution terminating for reason(s) noted above.;
        %goto bottom;
        %end;

    /* Macro variable merge and report */
    data _null_;

         if _n_ = 1 then put "Compare Macro Variables" / ;
         if eof & _changed = 0 then put "*** No changes detected ***";

         merge _delta_.macvarsPre(in=_pre)
```

```
            _delta_.macvarsPost(in=_post)
            end=eof;
      by name;
      length comment $20;
      if _pre & _post then do;
         if valuePre ^= valuePost then do;
            comment = 'Value changed';
            link print;
            end;
         end;
      if _pre & ^_post then do;
         comment = "Deleted";
         link print;
         end;
      if ^_pre & _post then do;
         comment = "Added";
         link print;
         end;
      return;

      print: _changed + 1;
             put name  comment;
             return;
 run;

 /* Datasets merge and report */
 data _null_;

      if _n_ = 1 then put "Compare Datasets" / ;
      if eof & _changed = 0 then put "*** No changes detected ***";

      merge _delta_.datasetsPre(in=_pre)
            _delta_.datasetsPost(in=_post)
            end=eof;
      by libname memname memType;
      length comment $100;
      if _pre & _post then do;
         if nVarPre ^= nVarPost then do;
            comment = catX(' ', "# vars changed from",
                           put(nVarPre,5.), "to",
                           put(nVarPost, 5.)) ;
            link print;
            end;
         if nObsPre ^= nObsPost then do;
            comment = catX(' ', "# observations changed from",
                           put(nObsPre,5.), "to",
                           put(nObsPost, 5.)) ;
            link print;
            end;
         if modatePre ^= modatePost then do;
            comment = catX(' ', "Mod datechanged from",
                           put(modatePre,dateTime16.), "to",
                           put(modatePost, dateTime16.)) ;
            link print;
            end;
         end;
      if _pre & ^_post then do;
         comment = "Deleted";
         link print;
         end;
      if ^_pre & _post then do;
         comment = "Added";
         link print;
         end;
      return;

      print: _changed + 1;
             put libname memname memType comment;
             return;
 run;

 /* Clean up after ourselves: deallocate temporary directory */
 libname _delta_ clear;
 %end;
```

```
       %bottom: %put End WHATCHANGED;

%lastStmt: %mend whatChanged;
```

## Examples of Use

Recall the output description of `%AEsummary` from “Why the Macro Is Needed,” above:

```
Output:   Datasets AE_SUMM and AE_DETAIL
          Global macro variables NPAT, NPAT_NOAE, NPAT_AE
```

If the header contains this documentation, then we can use `%whatChanged` to ensure that the only differences in the global macro variable table and the list of datasets are, in fact, the creation of these dataset and macro variables.

### Test Program 1: Identify Unwanted Items

During development of `%AESUMMARY`, attention was focused on the correct formation of output datasets and macro variables. Once development nears completion, `%whatChanged` can confirm that `%AESUMMARY` did not produce any “bonus” artifacts. The calling sequence is shown below:

```
%whatChanged(action=start)
%AEsummary(data=ae1, threshold=2, print=yes, obs=max)
%whatChanged(action=end)
```

An excerpt, below, from the SAS Log tells the developer that there is still some cleanup work to do. Macro variables `IDN` and `DSETCOUNT` and dataset `TRANSPOSE` may have been needed for producing the expected items. Once `%AESUMMARY` terminates, however, `IDN`, `DSETCOUNT` and `TRANSPOSE` represent problematic “leakage” that needs to be addressed.

```
Compare Macro Variables

IDN Added
DSETCOUNT Added
NPAT Added
NPAT_NOAE Added
NPAT_AE Added

Compare Datasets
WORK AE_DETAIL DATA Added
WORK AE_SUMM DATA Added
WORK TRANSPOSE DATA Added
```

### Test Program 2: Confirm Output is Correct

A variety of tools, some of which were described in “Other Tools,” above, can be used to eliminate the excess items produced by `%AESUMMARY`:

- `%SYMDEL` can be used to delete global macro variables `IDN` and `DSETCOUNT`
- The scope of `IDN` and `DSETCOUNT` could be made local
- `PROC DELETE` could delete dataset `TRANSPOSE`
- `%AESUMMARY` could store datasets in a subdirectory below `WORK`, using a separate LIBNAME. This is similar to the allocation method used by `%whatChanged` when `ACTION=START`.

Assuming these or other techniques were used to modify `%AESUMMARY`, we rerun the test case of the macro using the same calling sequence as before:

```
%whatChanged(action=start)
%AEsummary(data=ae1, threshold=2, print=yes, obs=max)
%whatChanged(action=end)
```

To no one’s surprise, the Log messages reveal the addition of exactly what was expected:

```
Compare Macro Variables

NPAT Added
NPAT_NOAE Added
NPAT_AE Added

Compare Datasets
```

```
WORK AE_DETAIL DATA Added
WORK AE_SUMM DATA Added
```

### Test Program 3: Get Unexpected Results

`%whatChanged` can also identify the *lack* of production of expected outputs. Suppose we run `%AESUMMARY`, this time changing the `THRESHOLD` parameter value from 2 to 0:

```
%whatChanged(action=start)
%AEsummary(data=ae1, threshold=0, print=yes, obs=max)
%whatChanged(action=end)
```

The Log messages, shown below, don't tell the developer "you're missing dataset `AE_DETAIL`." While it's likely that the omission was caught internally by `%AESUMMARY` diagnostics, the listing created by `%whatChanged` subtly notes the problem as well.

```
Compare Macro Variables

NPAT Added
NPAT_NOAE Added
NPAT_AE Added

Compare Datasets

WORK AE_SUMM DATA Added
```

## Comments

The purposes of this paper were to both describe a home-grown tool that supports good macro design and to encourage the reader to think of similar tools that he/she might need. `%whatChanged` was motivated by a macro "leakage" issue defeating the author's debugging efforts for hours. Here, as with any well-used tool, usage suggests enhancements for "Version Next." Among these:

- Compare more items (titles/footnotes, dataset variable attributes, system options)
- Give the user control over what's compared (any combination of macro variables, datasets, etc.)
- Clean up the output, maybe presenting results in a well-formatted PDF
- Save results in an output dataset.

Ways that `%whatChanged` can be enhanced are secondary to the more general theme of this paper: programmers who want to create reliable tools should always consider the need for utilities that extend what SAS offers and be willing to take the time to add to their developer toolbox. The toolbox has an indefinable "cool factor" and results in more macros that are reliable, robust, and have a professional look and feel.

## Contact

Contact the author if you have questions or comments: Frank@CodeCraftersInc.com

The CodeCrafters, Inc. web site has this paper, the PowerPoint slides used for the presentation, and all versions of `%whatChanged` available for download. The site also has other articles and resources related to macro best practices and using dictionary tables. Go to http://www.CodeCraftersInc.com.

## Fine Print