PharmaSUG China 2016 - Paper 18

Confessions of a Macro Developer – The trials and tribulations of building a large macro system

Rowland Hale, inVentiv Health Clinical, Berlin, Germany

ABSTRACT

Developing a robust, scalable, flexible and user-friendly macro library is a significant undertaking. This paper draws on recent practical experience of such a project and, without going into coding specifics, describes the major considerations, challenges and pitfalls encountered by the project developers along the way and how they were resolved. Areas discussed include project justification and stakeholder buy-in through to the collection of user requirements, development, validation, release and beyond.

Certainly, before the project can be given the go-ahead, the resource cost must be weighed up against the new system's potential benefits, and these can be significant. Major efficiency gains both for programmers and validators, as well as helping to ensure standards are adhered to or consistency maintained within a study, project or even companywide can be expected, and we consider how users are benefiting from the system we were involved in building.

More specifically, the paper covers topics ranging from initial design, parameter names and value checking, validation via test scripts and user acceptance testing, audit-proof documentation, handling post-release bugs and change requests, the benefits of adding automation to the validation and documentation processes (and how to do it using Excel as a central development database) and hurdles presented by migrating to a new environment. Working in an Agile programming environment is discussed as is best SAS® macro programming practice in general. Emphasis is placed throughout on user-friendliness, not only of the macros themselves, but also of the package as a whole. This includes the provision of training, user manuals and support from the development team, user-friendliness being an important aspect which contributes greatly to the overall success of the project; fundamental for the resulting system is that users see its benefits and want to use it!

INTRODUCTION

The author has been part of a small team involved in the development and maintenance of a complex macro system in conjunction with a client since close to the system's inception. The system we were involved in developing itself generates ODS graph templates, although what the system does is not directly relevant here – the paper is not about Graph Template Language (GTL), nor is it a text on the theory of project management or the software development process. It is intended more to record and share the experience of being involved in this project and the lessons learned along the way.

WHY DO WE NEED MACRO SYSTEMS?

A macro system is a set of related macros written for one or probably more of the following reasons:

- Carry out common tasks
- Abstract programmers from the need to write code requiring special knowledge (e.g: GTL)
- Facilitate the imposition of standards
- Enhance code readability
- Avoid possible common mistakes made by users
- Speed up program validation as system macros are pre-validated

It takes little effort to conclude from the above that we need macro systems to *improve efficiency*.

GETTING THE PROJECT OFF THE GROUND

Once the idea for a new system has come about, the relevant stakeholders will need to be brought in and presented with the system's potential benefits to the business. First and foremost, these should include major efficiency gains both for programmers and validators, as well as helping to ensure standards are adhered to or consistency maintained within a study, project or even companywide. This will require careful preparation and need to be backed up by hard evidence as the investment required is likely to be significant. Only when stakeholders have weighed up the resource cost against the potential benefits and been convinced of the latter will the project go-ahead be given.

The development process starts with requirements gathering from as many potential users as is realistic and will involve meetings with different user groups and possibly the creation of mock-ups if the system is to produce analysis outputs, for example. Although the requirements and resulting specifications should be as comprehensive as possible from the outset, it should be recognised that as programming progresses the requirements may well evolve as hurdles are overcome and new ideas come to light. Fundamental for the resulting system is that users see its benefits and want to use it. Official roles should also be established at this stage, and these include: Process Owner, Developer, Validation Lead, Tester, System User, Subject Matter Expert and possibly others.

DEVELOPING CODE

Wherever you work there will be programming standards and these should be strictly adhered to. We place great store on program layout and tend to think that if a program looks pretty then there's a good chance it works prettily too.

User requirements, including functional, performance and regulatory requirements if applicable must be turned into functional and technical specifications and a design which encompasses the many attributes of good software – efficiency, reliability, flexibility and scalability. There are lots of decisions to be made, not least about the structure of the system, including the division of the system into user and auxiliary sub-macros (auxiliary macros are system macros which contain task-specific code used multiple times by user or other auxiliary macros and which are not accessed directly by users). Furthermore the development team must be predisposed towards macro programming best practice and be agreed on programming standards at the start.

AGILE VS. WATERFALL

The waterfall model of software development implies a "downward" (hence waterfall), sequential flow through the various phases of the development process, from requirements gathering through development to release of the finished product. It generally assumes that all requirements can be defined and specified from the outset and that the project can reach successful completion based entirely on those initial requirements. Detractors of the waterfall model cite lack of flexibility and ability to adapt and evolve in the face of unexpected problems or changes in requirements or circumstance before the project is delivered, unhappy stakeholders who receive nothing until the entire project is delivered, a result which conforms to outdated requirements, and the potential for complete project failure should the project's budget be exceeded before completion.

The Agile approach, on the other hand, promotes flexibility and adaptability through numerous small iterations, known as "sprints", each of which results in a deliverable, working (and validated) software release increment. The Agile paradigm allows for constant review of requirements with new ideas and changes spawned from seeing released increments in action applied to subsequent increments. Compared to the waterfall model, transparency is much improved for the client, who receives regular releases of working and usable software, albeit not fully functional, early on, the software evolves as requirements evolve, and funds running dry before completion does not mean a fully wasted project with an incomplete and undeployable product at the end (see Ref: "The Agile Movement", "Principles behind the Agile Manifesto").

To date our project has applied the "not-quite-Agile" model. User macros have been released in batches, and there is an ongoing program of enhancement, new functionality and bug fixes, however our "sprints", whilst not quite marathons, have been too "middle distance" for comfort. There have been a number of reasons for this (one being members of the development team being distracted by other higher-priority projects), but it is certainly our aim to shorten future iterations – the advantages of doing so are clear to all. Our message here is: the more Agile you can become, the better.

ORGANISATION AND VERSION CONTROL

For better or worse, no dedicated version control software was used for the project. Versioning is carried out "manually" and the developers are relied upon to keep change histories up to date themselves (both in the macro headers and in a dedicated change history document). The development environment we use has a simple backup repository mechanism which is available for use at the developers' discretion. We are not necessarily recommending this approach, and it probably works thanks only to the small size of the development team. For systems with larger teams it has to be worth considering a version control system such as Apache Subversion (SVN) (see Ref: "Apache Subversion" and Rec. Read: Mengelbier, 2012).

SCALABILITY AND FLEXIBILITY

Giving full and proper consideration to building scalability and flexibility into the system from the outset can save some very big headaches further down the road. How this is done is beyond the scope of this paper and will depend very much on the purpose of the system. As an example, our client's ODS graphics system clearly does not implement the entire Graph Template Language, nor does it need to. But it does have the *potential* to implement very easily many more aspects of GTL than are currently offered. In a nutshell, because the system is based around

metadata set up in accordance with incoming parameter values, and there is a mechanism in place which loops through currently supported GTL options and which handles those options found in the metadata, extending support for additional options is achieved simply by having the macros write additional metadata and adding to the list of supported options. In addition, users can make use of one particular parameter which gives them access to the metadata, enabling them to customise the metadata in a way not provided for via the parameters. It is this sort of approach which promotes a system's scalability, flexibility and ultimate success as it grows and evolves.

BASIC MACRO PROGRAMMING GOOD PRACTICE

The following is a (non-exhaustive) list of Good Programming Practice recommendations for macros to be applied during development:

- Declare all local macro variables with %LOCAL (and ensure all declared variables are used).
- Consider implementing a naming convention for local macro variables – we used L_ as a name prefix.
- When setting system options within a macro, trap the options' current states beforehand and reset the options to those states at the end of the macro. For example:

```
/* At beginning of macro */
%LOCAL l_opts;
%LET l_opts = %SYSFUNC(GETOPTION(source,keyword))
              %SYSFUNC(GETOPTION(notes,keyword));

OPTIONS nosource nonotes;

/* At end of macro */
OPTIONS &l_opts;
```

- Avoid nesting macro definitions inside a macro.
- Name temporary data sets with a system-specific prefix and delete these at the end of the macro which created them (or as soon as no longer needed). An auxiliary "tidy-up" macro can be set up for this purpose.
- Ensure there is no redundant code which has been commented out (or which hasn't!) and no forgotten "debugging" %PUT statements present in validated and released macros.
- Ensure adherence to company and other agreed programming standards.
- Ensure consistency of code layout, in particular indenting of blocks and use of whitespace.
- Apply the KISS principle (Keep It Simple, Stupid!). This means avoiding code which is complicated for its own sake.
- Write appropriate comments *as you go*. The presence of comments will help not only others, but *you* when you come back to your code at a later date.

A macro which fulfils GPP not only makes for happier and more productive developers and but it also helps to inspire confidence in auditors.

PARAMETERS

Parameters form the primary user interface to a macro and therefore play a significant role not only in the macro's functionality but also in determining the success of the system from a user-friendliness point of view. There are various things to consider when setting up parameters, and we'd like to go through these in some detail.

PARAMETERS (INTRO)

Macros can contain a large number of parameters ranging from none through to fifty and more. Parameters in a system which contains multiple user macros can number in the hundreds. However, the number of parameters should be kept to a sensible minimum, so the first requirement for a parameter to fulfil is: its implementation must be justified! It is very easy to get carried away in the early days setting up different parameters for this, that and everything else, but stop and think – once implemented a parameter is to a large extent set in stone to be lived with by developers and users alike for years to come. A parameter can only be retired with the release of a new main version of the system, which usually happens only after several smaller releases have taken place over a period of time.

Parameters should be defined as keyword parameters. This allows default values to be defined (<parameter> = <value>) and helps to ensure clarity of macro calls. In some cases it may be desirable to allow a macro to accept a varying number of non-defined parameters and their values by specifying the PARMBUFF option on the %MACRO

statement. If doing so, incoming parameters must be carefully controlled by the macro via the SYSPBUFF automatic macro variable, and a suitable message issued to the log if the user defines something unexpected. Our client's system has an ODS style management macro which makes use of PARMBUFF to accept the large number of graph style elements as parameters. The macro contains a list of supported style elements in a macro variable and compares the incoming parameter list against this list, writing a warning to the log if an unsupported parameter is discovered.

PARAMETER NAMES

First and foremost, parameter names should be intuitive for the user and convey the parameter's intended use, so it is worth spending time, particularly for parameters which govern more obscure behaviour, thinking of suitably intuitive names, testing ideas against users if necessary. Do not shy away from long parameter names (you have up to 32 characters to play with) but only if this adds to the parameter name's intuitiveness. One example of a (justifiably in our opinion) long parameter name in the system we were involved in developing is: SPECIAL_PK_LEGEND_IGNORE.

Secondly, names should fully comply with a pre-agreed naming convention. If such a convention forms part of your company's programming standards, or one is already in place for other systems in use, then go with that convention, otherwise come up with a convention and stick to it. For example, you may wish to name "no/yes" parameters with an "_NY" suffix, and you may decide to separate "parts" of a compound parameter name with an underscore (as in the long parameter name above).

Thirdly, parameters which have the same or a similar meaning across macros within the system, or indeed across systems, should be given the same name. For example, you may wish to name parameters for input and output data sets INDAT and OUTDAT, but whatever their names, implementing them consistently makes life more predictable for the user. Several macros in our client's system have a parameter called STAT which defines the statistic to be displayed. Although the options for each of the macros are different – for one macro the options are MEAN and MEDIAN, for another SUM, FREQ and PCT – the general underlying meaning of the parameters is the same, so they have the same name.

Finally, there may be a requirement to implement a parameter for some specific behaviour for some specific circumstance. The initial temptation may be to restrict the new functionality to the specific circumstance defined, and to name the parameter accordingly, but before doing so consider whether it makes sense to implement a more generic solution, in which case the parameter name will need to be more generic too.

DEFAULT VALUES

Parameters should be given default values wherever it makes sense to do so and the values given should reflect the most commonly required scenarios, as well as company or other agreed standards. Of course this doesn't go for every system out there, but in ours, macros with around forty parameters produce standard outputs with users having to define just half a dozen parameters or fewer. Users would quickly lose interest in the system if they regularly had to go through large numbers of parameters setting the same values over and again. The remaining parameters are there for flexibility in terms of fine-tuning and customisation of outputs.

A number of our parameters have the default value <DEFAULT> (pointy brackets included). This has the overall meaning that default behaviour changes in accordance with how the user defines other parameters or depending on other circumstances. An example of this in the system we were involved in developing is the parameter CLASS_DATA = <DEFAULT>. This means:

- If the study's ADaM subject level ADSL data set exists AND it contains the *class* variable defined by the macro's CLASS parameter AND the *class* variable contains all *class* values in the *data* (input) data set, then set CLASS_DATA = ADSL.
- Otherwise set CLASS_DATA to the *data* (input) data set.

The detail here isn't important, the point being that default behaviour depends on what is going on elsewhere. Of course, to override the default behaviour the user can always set CLASS_DATA to another suitable data set.

When implementing conditional defaults like this, the conditional behaviour must be described clearly for the user in the user documentation (both in the user manuals and in the macro's header).

For default values, once again decide on a basic convention. Ours is to use YES or NO in full (as opposed to Y or N) and to use upper case for case insensitive values.

GLOBAL VALUES

Our client's system implements a mechanism whereby the user can set parameter values globally as global macro variables (this mechanism was inherited from another of the client's established systems and is in popular use).

These then override the default values of the parameters in question over multiple macro calls for as long as the global macro variables exist. To be treated as global parameters in the system we were involved in developing the macro variables must follow the following naming convention: <*system*>_PARAM_<*parameter name*>. As we are in the context of clinical data analysis, a common use for this is globally setting the CLASS parameter to the required treatment variable (the default value for the CLASS parameter is blank) e.g:

```
%GLOBAL <system>_PARAM_CLASS;
%LET <system>_PARAM_CLASS = trt01an;

/* Some nice code here including several macro calls */

/* Remove the global variable as soon as it is not needed! */
%SYMDEL <system>_PARAM_CLASS;
```

The mechanism is implemented such that the global value can be overridden in a given call simply by setting the parameter to a non-default value, and global values being used are written to the log. There is one gotcha here, and that is what happens if the compound global variable name <*system*>_PARAM_<*long parameter name*> becomes longer than 32 characters. This leads to a SAS error and is one we don't attempt to trap seeing as the few parameters with names long enough to cause problems are unlikely to have any need for use globally, and SAS programmers should themselves be aware of the 32 character limit when setting up the global variable.

PARAMETER VALIDITY

One aim of a macro system should be to prevent an uncontrolled system crash if the user enters an invalid parameter value. Achieving this means the implementation of extensive parameter checking at the beginning of each user macro, and aborting the macro in a controlled fashion, and with an informative error message written to the log, should any parameter value fail its validity check. We started by setting up a separate auxiliary macro to handle the parameter checking for the various user macros, but even in the early days when there were just five user macros, the check macro quickly became unwieldy and difficult to maintain with many of the checks being conditional upon the calling macro. The first major problem came when "general" checks (checks for parameters present in all user macros) became conditional because the parameters weren't present in a newly implemented user macro. At that stage the decision was taken to move to an Excel based solution, and despite the initial setup cost the benefits have been reaped ever since (more information about the Excel based automated solutions we have implemented follows below).

Purely for reasons of simplifying the validation of parameter checking during system validation, the decision was taken to abort the macro on the first encounter of an invalid parameter value. This can lead to a user being unhappy about learning of his or her many parameter definition errors one at a time! However in practice, this has not been problematic and the few comments about this received by the development team have been successfully handled with a suitable explanation of how the system is validated. Aborting a macro is handled using %GOTO as follows:

```
%IF &parameter_error = 1 %THEN %GOTO ABORT_MACRO;
    /* Main body of macro here */
%GOTO END_MACRO;
%ABORT_MACRO:
    /* %PUT helpful message for user here */
%END_MACRO:
    /* General tidying up done here, whether macro is aborted or not */
```

Parameter values should always be case insensitive unless case sensitivity is significant (such as in text labels). Where appropriate, parameter values can be up-cased once at the top of the macro to avoid having to do this every time the value is tested in an IF condition:

```
%LET stat = %UPCASE(&stat);
```

Consider too the use of aliases for certain values, the most obvious ones being Y for YES and N for NO. A nice way to test for YES in a variable once post validity checking has been carried out is shown below. The test is case insensitive catching all of Y, y, YES and yes. YABADABADOO would also pass the test, you cry, but that wouldn't have passed its initial parameter check, so all is well.

```
%IF %SYSFUNC(FINDC(&title_ny, y, i)) = 1 %THEN %DO;
```

Checking for a blank parameter, it turns out, is not as straightforward as it may seem (see Rec. Read: Chung, 2009 for a comprehensive study of this question). Our client's system relies on testing the length of a parameter and deems it blank if the length is zero. To date we have encountered no problems from applying this test to validated parameters:

```
%IF %LENGTH(&class) > 0 %THEN %DO;
```

Table 1 below shows a selection of common parameter validity checks:

Parameter type	Check that…
Data set	Data set exists
	Data set contains data
Variable	Variable exists in data set
	Variable is of expected type
Data	Variable contains only non-missing values
	Variable contains no negative values
Discrete list of options (e.g. for yes/no parameters)	Parameter value is one of YES Y NO N yes y no n
Numeric	Value is integer
	Value falls within expected range
General	Parameter is not empty
List	All list items are valid
Dependent	Parameter A is filled when Parameter B is filled

Table 1. Common parameter validity checks

SPECIAL CHARACTERS

Particular care has to be taken to handle special characters in parameter values in the way that the user expects, especially & and % in free text parameters. Warnings like these are to be avoided:

```
WARNING: Apparent symbolic reference VAR not resolved.
WARNING: Apparent invocation of macro TEST not resolved.
```

This means appropriate use of macro quoting functions (see Rec. Read: O'Connor, 2003 and Patterson & Remigio, 2007).

VARYING VALUE TYPES

It may be considered desirable for certain parameters to accept both constant values and variables. An example of this is the parameters we have for displaying reference lines on graphics. These parameters accept a list of zero or more constant values and/or variables, the latter being useful, for example, for displaying laboratory test dependent lower and upper reference limits defined say in the ADaM laboratory data set ADLB: YREFLINE = ANRLON # ANRHIN. Depending on the parameter's role, a strategy for handling an assigned variable's values must be specified (for example, whether to take the minimum, maximum or mean value of grouped data, can missing values be present, etc.).

SYSTEM VALIDATION

Non-negotiable is that the new macro library be properly validated and documented – these activities are of vital importance, form a significant portion of the project in terms of resource and should be factored in from the very outset. Unfortunately for the developers, validation and associated documentation takes significantly more time than program development.

WHAT IS VALIDATION

In general terms, software validation is the process by which software is formally tested to ensure it works in full accordance with its specification and intended purpose. Software testing is a large topic, and it is beyond the scope of this paper to go into the relative merits of the different software testing techniques. Common ways to validate SAS programs in a clinical programming environment are double programming and code review. Whilst double programming may be appropriate for testing specific aspects of a standard macro library, for example where complex derivations are involved, a third method, namely a series of formalised test scripts, will be required to ensure that the macros fulfil their requirements and do so in robust fashion. The process and its requirements will likely be described in company SOPs.

The validation of our client's system is made up of a series of test scripts which implement unit, integration and regression testing. Although our client's system comprises not only user macros but also a significant number of

auxiliary macros not accessed directly by the user, only two specific areas undergo unit testing – parameter validation being one. The vast majority of the validation relies upon integration testing, that is, top-down testing of the user macros with a variety of data and an extensive set of parameter setting combinations. With the validation resource at our disposal, this integration vs. unit testing approach was considered an acceptable compromise, and this has been borne out by the resulting robustness of the released system over a period of time, although for other systems it may be considered necessary to carry out unit testing of all auxiliary macros. With multiple user macros containing forty or more parameters, it must be expected that test cases for the entire system will number not in the hundreds but the thousands.

REGRESSION TESTING

We cannot stress enough the value of regression testing as part of the validation process. Regression testing is the process by which outputs are compared back against a library of validated "reference" outputs from earlier runs of the test scripts. Regression testing goes a long way towards protecting against unwanted and unforeseen side-effects of changes to the code as enhancements, bug fixes and other changes are implemented and the macros evolve through their various versions and releases. With each new feature implemented or bug fixed come new test cases and new outputs for the reference library – and subsequently increased confidence that changes to the system have had no adverse consequences. Note though that intended changes to existing behaviour can render a significant number of the reference outputs invalid in which case these need to be replaced with the carefully validated new outputs.

Automated regression testing is set up relatively easily by making good use of the COMPARE procedure and its return codes ("Base SAS 9.2 Procedures Guide: Results: COMPARE Procedure").

LOG CHECKING

Having a mechanism in place to check the logs for expected and unexpected messages is also an important aspect of the validation process. There will probably be test cases in place which test for the correct issuing of error and warning messages, for example in test cases which test the behaviour of a parameter validity check when the parameter is fed an invalid value. The appearance of an error or warning message in the log does not then signify failure of the test case if the message is expected. By the same token, unexpected messages must also be checked for so that the result of the test case can be deemed unsuccessful if any such messages appear.

USER ACCEPTANCE TESTING

User acceptance testing (UAT) has been something of a moot point for us. There is no doubting its value, not least because users have a totally different perspective of the software from that of the developers and they can provide new insight, novel ideas and valuable constructive criticism. The problem for us was finding willing, able and available resource for the task in hand. Users, in the form of other SAS programmers, do not generally bask in the luxury of sitting around waiting for a nice UAT task to come along. Most are overloaded with study work and stressed by tight deadlines, so UAT can be an unwelcome addition to an already overfull plate. And even with spare time in hand, UA testers vary in their levels of engagement and fastidiousness. One of our macros did benefit from a particularly diligent UA tester though, which only confirmed how worthwhile UAT can be if done properly.

DOCUMENTATION

Few programmers relish the prospect of writing documentation. Documentation starts with the macro program itself - not only should the code be self-documenting insofar as that is possible through the use of appropriate data set and variable names, a reader-friendly layout and a logical structure, but also the header box should contain parameter descriptions with information about valid values, examples of macro use and a change history. The code should of course be appropriately commented too – not only about what's going on, but why.

Beyond the programs themselves, documentation comprises detailed program and test script specifications as well as validation reports, all properly reviewed, versioned and signed off in accordance with SOPs, as well as comprehensive user manuals with examples, particularly important for more complex systems.

AUTOMATION

Our project makes copious use of Excel as a central development database for parameter descriptions and the definition of parameter checks and test scripts. Whilst this approach may not be appropriate for smaller projects because of the initial setup overhead, we benefited greatly from having single sources of description and definition data as well as the ease of use that Excel affords compared to maintaining, for example, 10K+ lines of SAS code in one test script (of many).

The larger a system becomes the harder it is to maintain (obvious really). It became clear to us fairly early on that several areas within the project were becoming overly difficult to manage, for a number of reasons:

- Ever increasing volume of code (particularly test scripts)
- Repetitiveness of code, but only partially so (not repetitive enough to permit “macroising”) – e.g. parameter value checks
- Ongoing reduction of code genericity as more user macros released – e.g. parameter value checks

If an aspect of the project is sufficiently systematic and self-contained in nature it may well lend itself to being automated, or in other words, to being data or metadata driven. By this we mean making use of SAS to generate SAS code based on a data source external to SAS. Whilst SAS can read a wide range of external data formats, there is probably only one which is properly suited to this purpose on Windows – Excel. Advantages of using Excel as a SAS program definition database are various:

- Ease of data entry
- Good overview of data
- Easy to copy and paste data whilst maintaining overview
- Colours and other ways to format data facilitate review and maintenance
- Column filtering of data
- Familiarity for SAS programmers

An Excel file is easily read into a SAS data set using the IMPORT procedure (see section “How to generate a SAS program from Excel” below).

PARAMETER VALUE CHECKING

As previously mentioned, we started with a single, but continually growing, parameter check macro which was called near the top of each user macro. It contained a large number of parameters, in fact, with the exception of a few free text parameters not subject to value checking, all of the parameters of all of the user macros. Endless conditions were necessary to determine which check was relevant to which parameter from which user macro. The macro was rapidly becoming less and less pretty and very difficult to maintain when new user macros were added to the system or new parameters added to existing user macros. The question whether or not to automate had become a no-brainer. Finally the bullet was bitten and over the course of several days the Excel file was designed, check definitions transferred into the file and a SAS program to read the file and generate multiple parameter check macros (now a separate check macro for each user macro, each with user macro specific parameters) developed.

The Excel file contains three sheets:

- Main sheet with list of all checks plus “non-custom” check definitions. Columns include parameter name, check name, check relevance (list of user macros which contain the parameter for which the check is relevant), check conditions, message type (error or warning), and message on failure.
- “Custom” checks too complex to define via the columns on the main sheet. Columns include parameter name, check name and custom checks in the form of SAS code – which is copied directly by the generation program into the parameter check macro. It isn’t ideal to have non trivial SAS code inside an Excel sheet, but it had to go somewhere and to keep the definitions in a single location it was decided that this was the least of several evils. At the time of writing, 18 of nearly 200 checks defined are custom checks.
- Change history – used by the generation program to populate the change history section of the parameter check macro header.

The initial setup overhead justified itself very early on. We now have a series of neat, self-contained, user macro specific parameter check macros which are maintained exclusively and easily via the Excel sheets. Because the macros are generated automatically, code layout is consistent throughout. And the users benefited too – when setting up the check definitions we took the opportunity to add an additional column containing a hint for the user, written to the log by the parameter check macro on failure of the check, about how to resolve the error. Not only easily implemented, but without the Excel automation the idea would probably not have materialised.

USER MANUALS

Good quality user manuals written in understandable language are vital for the overall user experience. Our user manuals contain a description of what the macros do, a list of parameters in tabular form with valid values, default values and a short description, the macro call as SAS code showing default values, a list of parameters in text form with valid values, default values and a long description, errors handled and associated behaviour, and a set of real-world examples with outputs. If user manuals can be written or reviewed by a native speaker, so much the better.

The user manuals are generated automatically, the basis being – you guessed it – an Excel spreadsheet. The principle is exactly the same as before with columns this time containing parameter names, the user macros the parameter belongs to, default values, valid values, short descriptions and long descriptions.

In addition to the benefits of an Excel based approach already mentioned, defining and storing “user manual” information in a single location means a “one stop maintenance shop” for information which is then available for writing out by “generation” programs to multiple locations such as user manuals in Word and HTML format and user macro headers. In the case of the latter, a SAS generation program reads in each user macro program as text and writes it out again inserting into the header a list of parameters (together with valid values, default values and short descriptions) read in from the user manual Excel sheet. The generation program knows where to insert the parameter info thanks to tags present in the initial header (and kept there):

```
* <parameters>;
  /* Parameter descriptions inserted here */
* </parameters>;
```

TEST CASES

We gained huge benefit from applying the Excel based automation approach to setting up test scripts. Test scripts for complex macros are long and tedious to set up, but being generally systematic in nature, they lend themselves very well to being defined in Excel. Each user macro has its own test script and because functionality of some of the user macros is similar, certain areas of these user macros’ test scripts are similar too, but keeping track when copying and pasting multiple chunks of program between test scripts comprising in excess of 10K lines of code is, to put it politely, challenging. With Excel, things are a whole lot easier and programmer sanity is preserved (insofar as that is possible). The test scripts still contain their 10K+ lines of code, but these are maintained exclusively via the Excel definition sheets (one sheet per user macro), where copying and pasting is now a practical proposition and the overview is maintained, much facilitated by the ability to filter columns.

Test cases which separately test individual options from a discrete list of valid options for a particular parameter can be defined in the Excel sheet with the help of a designated column which contains a complete list of the options and/or a special list syntax. The generation program then knows to loop through the list, creating a new test case for each listed option as it goes. In this way a single row in the Excel sheet can easily define multiple related test cases. The list syntax which we have employed in the Excel looks like this:

TITLE_NY = YES§Y§NO§N

This results in four test cases being generated, one for each of the listed options delimited by the § character.

VALIDATION DOCUMENTATION

It is highly desirable to have an automated system for generating validation documentation in place. We benefited from having at our disposal a macro (with Java in the background) which takes a data set as its only parameter and generates a Word document based on a template (another Word document) and mapping information contained in the data set. The template document contains fields and text tags (defined by us) which are populated and replaced in accordance with the mapping information. Word tables in the template are defined with a header row and single data row containing a text tag – so two rows in total – and are expanded automatically by the macro to accommodate all of the rows present in the mapping data set defined with the corresponding text tag.

Having a 21 CFR Part 11 (see Ref: "Part 11, Electronic Records; Electronic Signatures — Scope and Application") compliant document management system at your disposal too is also highly desirable, particularly if document signatories are located across multiple sites. This greatly simplifies the collection of approvals and avoids the need for repeated scanning and emailing of documents signed in wet ink.

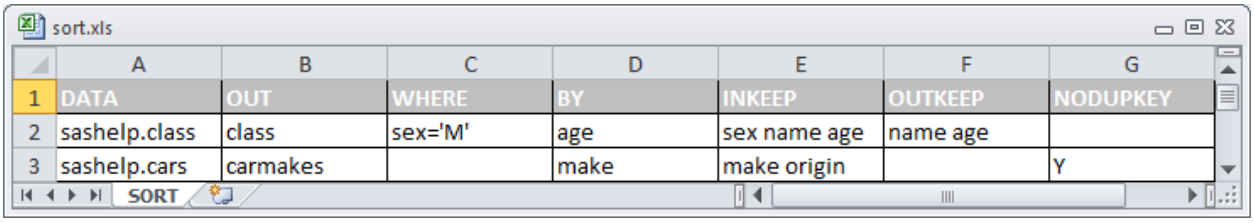
The further you can go with automating the generation of documentation the better, and the benefits really speak for themselves particularly, say, when a properly documented hot fix has to be released in double-quick time.

HOW TO GENERATE A SAS PROGRAM FROM EXCEL

Generating a SAS program from Excel using SAS is not difficult. The steps are as follows:

- Set up Excel sheet with appropriate column names, and populate as required.
- Read Excel file into a SAS data set using the IMPORT procedure.
- Generate SAS program using a FILE statement and PUT statements within a _NULL_ DATA step.

For illustrative purposes, the example below shows how SORT procedures can be defined in and generated from Excel. Display 1 shows an Excel sheet with columns representing various SORT procedure options:


sort.xls

	A	B	C	D	E	F	G
1	DATA	OUT	WHERE	BY	INKEEP	OUTKEEP	NODUPKEY
2	sashelp.class	class	sex='M'	age	sex name age	name age	
3	sashelp.cars	carmakes		make	make origin		Y

SORT

Display 1. SORT procedure definitions in Excel

The Excel sheet is read into a SAS data set called “sort” using PROC IMPORT:

```
PROC IMPORT DATAFILE='c:\temp\sort.xls'
            OUT=sort
            DBMS=xls;
    SHEET='SORT';
RUN;
```

SAS programs are plain text files, so generating the program with the SORT procedures is now just a matter of defining the output file using the FILE statement in a _NULL_ DATA step and writing the SAS code to the output file as text using PUT statements, as below:

```
DATA _NULL_;
    FILE 'c:\temp\sort.sas';
    SET sort;
    PUT 'PROC SORT DATA=' data +(-1) @;
    IF NOT MISSING(inkeep) THEN PUT ' (KEEP=' inkeep +(-1) ')' @;
    PUT ' OUT=' out +(-1) @;
    IF NOT MISSING(outkeep) THEN PUT ' (KEEP=' outkeep +(-1)')' @;
    IF nodupkey = 'Y' then put ' NODUPKEY' @;
    PUT ';';
    IF NOT MISSING(where) THEN PUT @5 'WHERE ' where +(-1) ';';
    PUT @5 'BY ' by +(-1) ';';
    PUT 'RUN;' /;
RUN;
```

Output 1 shows the generated SAS program *sort.sas* with the two SORT procedures as defined in the Excel sheet:

```
PROC SORT DATA=sashelp.class (KEEP=sex name age) OUT=class (KEEP=name age);
    WHERE sex='M';
    BY age;
RUN;

PROC SORT DATA=sashelp.cars (KEEP=make origin) OUT=carmakes NODUPKEY;
    BY make;
RUN;
```

Output 1. Generated SAS program *sort.sas*

RELEASE AND POST RELEASE

Once validation is complete and the system has been signed-off for release, the system can be released for production use. There are important questions regarding the release, which hopefully will have been considered in advance – where will the production system be located, how will users access it, and how will they know how to use it?

The first question is whether you are releasing compiled macros (in a catalog) or the original source macro statements as SAS programs. For us the answer was the latter (to learn more about storing compiled macros please see the SAS online documentation ("SAS 9.2 Macro Language: Reference, Saving Macros Using the Stored Compiled Macro Facility")).

The second question is where to store the released system. The answer has to be a read-only location standardised for macro systems in your organisation i.e. a specific pre-designated systems area with standardised folder structure, if applicable allowing for previous system versions to remain available when a new version is released.

EFFECTING THE RELEASE

Effecting a release involves:

- Copying the macros (or catalogs) to the production systems area.
- Making user documentation available.
- Sending a nicely designed (colour helps!) email to users and other stakeholders giving details of the release (new macros, changes etc.) and user documentation, examples if applicable, and acknowledging all those who sweated blood and tears to bring about this fantastic system (developers, testers, advisers etc.). Attached to our release emails is a "Getting started with…" PowerPoint presentation giving more detailed information about the release. This is also a good opportunity to solicit feedback from users and state how important such feedback is for the continued enhancement of the system.

TRAINING AND SUPPORT

A system can only deliver its full potential if users know how to exploit the system's full potential. We offer the following training methods and information sources to impart knowledge about the system:

- Release emails which provide an overview of new macros, enhancements and bug fixes included in the release together with examples if applicable.
- A "Getting started with…" PowerPoint presentation which gives more detailed information about the release.
- User manuals which provide detailed information about each user macro and its parameters.
- Macro headers which contain abbreviated information about parameters and valid options.
- User and super user training delivered face-to-face or remotely. Super users for us are client nominated users, one per site, who receive comprehensive training and who are intended to be the first point of contact when system related questions arise.
- Training new employees on the system as part of their onboarding process.

In addition, members of the development team are always on hand to answer user questions.

CHANGE REQUESTS AND BUG FIXES

Bugs and change requests are inevitable aspects of system maintenance. Important considerations here are having a centralised system for entering and tracking bugs and change requests (your email inbox will not do!), properly categorising bugs and change requests in terms of priority, and having a defined accelerated process in place for fixing bugs and validating, documenting and releasing the fixes. Whilst it may be commendable to implement third-party software to manage bugs and change requests, the least that will suffice is a centrally accessible Word document set up for this purpose. In preparation for each system release we have an Excel task list which lists and tracks each change request, bug fix and enhancement approved for the release. Against each item is a series of subtasks (implement, add test case, update manual etc.) which are ticked off as completed.

MIGRATION

Migration to a new version of SAS or to a new operating environment brings with it a new set of considerations and additional resource requirements and can result in the need for more than one production version of a system.

When migrating to a new version of SAS will the previous version of SAS be retained for production use? If so it may well be required to maintain multiple versions of the system built respectively on the different SAS versions (say Version A for SAS 9.2 and Version B for SAS 9.4). In particular this has implications for validation as Version A and Version B will need to be validated separately from this point on. Apart from the application of critical hot fixes, Version A may well be "frozen" whilst Version B advances as development continues and the new SAS version's capabilities are taken advantage of.

Whilst our client's system has not yet progressed to a new version of SAS, it has experienced migration to a new operating environment. This means, inevitably, maintaining the system for the two environments, at least until the old environment is phased out. Testing the system on the new environment threw up a (thankfully) small number of (obscure) incompatibilities, however it was possible to resolve these in such a way that the system subsequently passed validation on both environments. This is of course the ideal scenario, as maintaining and validating two diverging versions of the same system on different environments is far more resource intensive.

CONCLUSION

A well thought-out and properly implemented macro system can bring significant benefits, in particular greatly improved efficiency and consistency of outputs. Our client's system enables users to create complex client-standard and customised ODS graphics quickly and easily without knowledge of the statistical graphics procedures or Graph Template Language and the many options of both, often by setting just a few parameters. The success of any system depends not only on the system fulfilling requirements, but also on users *wanting* to use the system, and this in turn depends on the system meeting the needs of the user as well as its flexibility, robustness and overall user-friendliness. For larger systems, automation can be given a significant role and this can greatly facilitate development and maintenance of areas of the project which are systematic in nature.

REFERENCES

Apache Subversion. *Subversion.apache.org*. Retrieved from: https://subversion.apache.org

Part 11, Electronic Records; Electronic Signatures — Scope and Application. *Fda.gov*. Retrieved from: http://www.fda.gov/RegulatoryInformation/Guidances/ucm125067.htm

Principles behind the Agile Manifesto. *Agilemanifesto.org*. Retrieved from: http://agilemanifesto.org/principles.html

The Agile Movement. *Agilemethodology.org*. Retrieved from: http://agilemethodology.org/

Base SAS® 9.2 Procedures Guide: Results: COMPARE Procedure. Support.sas.com. Retrieved from: http://support.sas.com/documentation/cdl/en/proc/61895/HTML/default/viewer.htm#a000146743.htm

SAS® 9.2 Macro Language: Reference, Saving Macros Using the Stored Compiled Macro Facility. *Support.sas.com*. Retrieved from: http://support.sas.com/documentation/cdl/en/mcrolref/61885/HTML/default/viewer.htm#a001328775.htm

RECOMMENDED READING

O'Connor, S. (2003). Secrets of Macro Quoting Functions – How and Why. In *Twelfth Annual Northeast SAS User Group Conference*. Washington, DC. Retrieved from: http://www.ats.ucla.edu/stat/sas/library/nesug99/bt185.pdf

Patterson, B. & Remigio, M. (2007). Don't %QUOTE() Me on This: A Practical Guide to Macro Quoting Functions. In *SAS Global Forum 2007*. Orlando, Florida. Retrieved from: http://www2.sas.com/proceedings/forum2007/152-2007.pdf

Mengelbier, M. (2012). Simple Version Control of SAS® Programs and SAS Data Sets. In *SAS Global Forum 2012*. Orlando, Florida. Retrieved from: http://support.sas.com/resources/papers/proceedings12/365-2012.pdf

Chung, C. (2009). Is This Macro Parameter Blank?. In *SAS Global Forum 2009*. Washington, DC. Retrieved from: http://changchung.com/download/022-2009.pdf

Carpenter, A. (2004). *Carpenter's complete guide to the SAS macro language*. Cary, NC: SAS Institute.

CONTACT INFORMATION

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

Name:	Rowland Hale
Enterprise:	inVentiv Health Clinical
Address:	Joachimsthaler Strasse 10-12
City:	10719 Berlin
Country:	Germany
Email:	rowland.hale@inventivhealth.com
Web:	inVentivHealth.com/Clinical