# Infrastructure Designed to Maximize Workflow

Paul Hamilton, Omeros Corporation

All clinical statistical programmers work within a common set of constraints: we must be prepared to support the production of Clinical Study Reports after a trial is complete; we must be prepared to support filings to various regulatory agencies; we must operate under the aegis of CFR 21 Part 11; we must contend with frequent changes in specifications and contracting timelines; and most of us use SAS™ software in our work. This paper presents an architecture purporting to maximize programmer workflow, minimize manual steps, and optimize programmer efficiency. Guiding principles include: single source of metadata, stored locally; all processes driven by metadata and automated through software; hands-off production of finished publishable products; and minimizing human-only tasks wherever possible.

## Introduction

After working in a variety of professional situations (mid-level CRO, startups, mid-level pharma and large biotech) the author had the opportunity to build the infrastructure for a small company from the ground up. This was a chance to address many of the tool and process issues that impeded workflow or created unnecessary complications while adding little to no value. This paper describes the resulting infrastructure; while very much a work in progress, the results have been gratifying and reduced job stress considerably.

## Mechanics

SAS executes on a server running under Windows Server. All coders VPN into the server, and typically develop code in SAS Display Manager System (version 9.4M3). All SAS resources (code, datasets, format libraries, template stores, style templates, user-defined functions, and macro libraries) are stored under folders on a local drive. Having contractors use their own Windows or Mac/OS machine means they can onboard as quickly as IT can set up an account and enable access to the system. Having one copy of SAS to maintain minimizes redundancy and simplifies continual efforts such as applying SETINIT or Hot Fixes.

## Customization

There are three layers to customizing the SAS System: The main configuration file (!!SASroot \SASFoundation\9.4\nls\en\sasv9.cfg); the Autoexec file(s), and Windows shortcuts. We have not had occasion to modify the Configuration file as provided by SAS Institute during the install process; all localization takes place through using Autoexec and shortcut icons pointing to appropriate folders.

### Folder Structure

Our within study folder structure is standard: \Tables \Listings and \Figures are obvious, each containing `\code` and `\output` folders. We also have study-level folders \docs (Protocol, SAP, Table shells, any supporting documents for the study), `\macros` (project level macros; rarely used), `\projmeta` for project level metadata (much more about this later). An `\explore` folder holds any reports or data transmissions requested during or after the study but not supported by the SAP. Project-critical e-mails are printed into PDF documents in the `\docs` folder.

### Project Initiation

All of these folders are created in the DEV environment (then copied over to PROD just before database lock and unblinding). A tool to automatically create the study folders would be helpful, but as it only takes a few minutes to set up (and rarely), automating this process is of low priority.

Inside each `\code` folder, the first file created is [`autoexec.sas`], always named exactly that. Many workplaces require the first line of executable code in any project file to be something like:

```
%include "\study\folder\stage\code\setup.sas";
```

This is the first place our system diverges from "normality". The SAS system supports the concept of a "magic file" containing codes that one wants to have automatically executed upon initiation of a SAS session (either interactive or batch). This has been in place since at least the early 1980s, starting on the mainframe and continuing into every directory-based OS (e.g. not TSO/JCL based systems). Why generations of SAS coders have ignored this and name their "magic" files by other names is a conundrum. The penalty for not using this simple system is severe: *every single* SAS code must be altered to change the stage from {DEV -> PROD} when DB lock and unblinding occur. Failure to do means that the `libname` statements points back to the locked database and random (not actual) treatment assignment codes for subjects. The results of this oversite can be catastrophic.

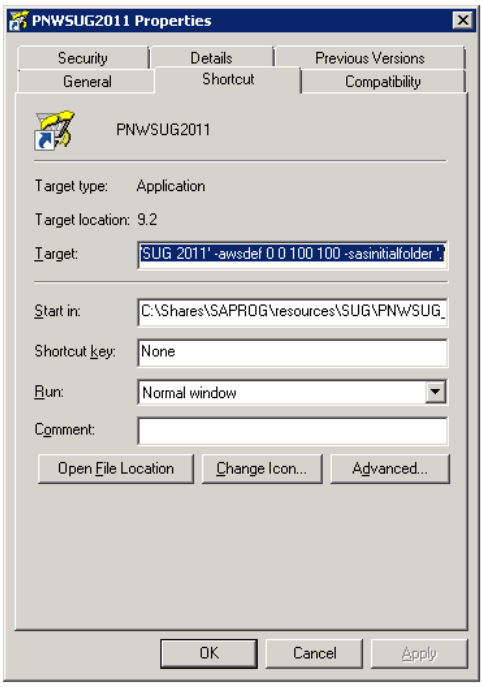
Even if an organization does have a way of checking to ensure that all required pointers have been updated in the move to production, it still takes considerable time to make the edit changes. If one instead uses the natural autoexec method, then one merely has to copy over a handful (SDTM, ADaM, T, L, F) of autoexec files from DEV to PROD, and make the changes in those files. This can be done at any time before the actual DB Lock — Production move, allowing plenty of time to QC the setup.

**Windows Considerations**

This section assumes that you are coding SAS using the old-fashioned Interactive Development Environment (IDE), namely Display Manager System (DMS). If instead you use an external editor to code, and run SAS in batch mode, or use SAS Enterprise Guide or SAS/BI products to access the power of SAS then this section is irrelevant.

Little of the proposed architecture is influenced by choice of operating system. However, usage of visual shortcuts under Windows makes life just that little easier. The name applied to the shortcut could be a path e.g. Product-Study-usage (usage in {TLF etc.}) or the Product – Study information could be contained in subfolder names leaving just usage for the shortcut name.

The second box [Start in] just contains the Windows path to the folder where the associated code lives, i.e. …\Study\ ENV\Tables\code. The contents of the first box [Folder] are dissected below.




The first and most critical part of the [`Target`] box is a quoted string holding the location of the SAS executable on your system. This will typically be on some locally attached drive, and end in [`...\sas.exe`]. If you build a SAS shortcut with just this specified, then every time you launch SAS, your "current folder" will be in the same place, probably under your [`\user`] folder. If you are required to store SAS code in particular product / study / TLF folders this is very inconvenient, as you will have to manually change folders for every SAS session. Luckily there is a simple solution: place the particular study folder (i.e. [`\PROD123\DEV\Study01\Tables\Code`] ) inside the 2nd [Start in] box, and in the [Target] box add the SAS option [`-sasinitialfolder '.'` ]. Now when you double-click the shortcut icon, SAS will automatically move to the correct study folder, initiate a DMS session, look around for an [`autoexec.sas`] file (which you will have provided of course), execute the various LIBNAME and options present in that file, then present you with a clean Program Editor to begin your work.

Other options which can be included in the [Target] box include:

| | |
|---|---|
| `-nodmsexp` | Turns off the small SAS Explorer windows along the left side. |
| `-awsdef 0 0 100 100` | Uses 100% of the screen for the SAS application. Saves having to hit the Maximize button when opening a session. |
| `-awstitle 'text'` | Some text to ID what project you are working on, i.e. [Study123 Tables]. Appears in the top left border of the SAS session |

Setting up one of these shortcuts might take a few minutes in the beginning. Once you have the first one working the way you like, just select it, <Copy> <Paste> <click-rename> to build a new one. Then just replace the [Start in] box with the folder path for the new project, and if you use the [`-awstitle`] command change the text to reflect the new project – a matter of one minute.

## Project Metadata

The typical 3-level system for SAS resources (format catalogs, macros, styles etc.) needed in a project was employed: generic resources are located high in the folder hierarchy; Project level resources are stored in the local project folder, and individual program code can override these with entries created in the [`work`] library. There have been myriad papers written over the last three decades about how to set this up, and the method utilized is completely ordinary in this respect.

With one exception: Since the pharmaceutical business is a regulated industry, we typically have very formal documents (Protocol, SAP, and Table Shells) that carefully pre-specify a number of reports for us to build: Tables, Listings, and Figures (hereafter referred to as *TLFs*). Most organizations appear to have a common workflow of keeping all project-related metadata (Table #, Patient Populations, Titles / Footnotes etc.) in a project-level Excel spreadsheet. We employ the same (plus more, to be discussed) information, but choose instead to store it in ordinary SAS data sets. By convention, each study sub-folder has a *libref* for [`projmeta`] contained in the [`autoexec.sas`] file. The project-level metadata consists of mainly two items: a SAS data set called [`projmeta.projman`], and the project-level format library [`projmeta.formats`]. The ProjMan data is keyed on Program name [`projmeta.projman.ProgName`].

As mentioned above, the first line of executable code in our programs is not a [`%include …`] statement, but rather an invocation of a standard enterprise-wide macro [`%getPgmName()`]. This macro combines the functionality of two macros from the SAS web site [www.support.sas.com] and simply creates a global macro variable in the current (interactive or batch) session containing the name of the program being executed. Thus every program is self-aware of "who it is" without the programmer burden of hard coding a [`%let pgmname=XXX;`] statement (and avoids a potential disaster if a program is copy-paste-rename-tweaked while forgetting to re-tweak the hard coded name).

### Report Structure

The relationship between one discrete SAS program and its potentially many output objects is complex. Probably most cases fall into one of the entries below:

| | |
|---|---|
| 1-1 | Execution of one SAS code (i.e. Demog.sas) results in the creation of one output object (e.g. Demo.rtf) |
| 1-? (by) | One SAS code will produce a data-driven number of separate outputs, driven by the number of *by*-values in a category variable. However the number of *by*-values is predictable from study structure (e.g. one separate report per Study Site) |
| 1-? (random) | One SAS code might produce multiple output objects based on pre-defined values of data, but the number of *by*-values can fluctuate up until Database Lock. |
| 1-*fit* | Sometimes one code execution produces only one type of output but that is voluminous enough that it will never fit on one report page (e.g. Laboratory summaries over time and over multiple analytes). The code must produce enough discrete reports to allow all output to *fit* within publishing guidelines. |

These are common issues within our industry and all organization have developed techniques (often-involving report macros) to address the problems. An analogous issue occurs even within a 1-1 report construction: Most tables (and some Listings) are segregated into separate sections; some may be slight variations on each other while others can be wildly different.

Assuming the simple example of 1-1, a slight variation on how table construction occurs was developed. We view each Table (and some Listing) report shells through an imposed structure of (up to) three "meta" variables that define the table structure:

| | |
|---|---|
| Page # | `pagenum` |
| Panel # | `panel` |
| Row | `row` |

Consider the following generic Table shell:

| Parameter<br>Category/ Statistics | Group A<br>(N=XX) | Group B<br>(N=XX) | Total<br>(N=XX) | *meta* |
|---|---|---|---|---|
| Age (Year) N | xx | xx | xx | *Page 1* |
| Mean (SD) Median | xx.x (xx.xx) xx.x | xx.x (xx.xx) xx.x | xx.x (xx.xx) xx.x | *Panel 1* |
| Min, Max | xx, xx | xx, xx | xx, xx | *Rows 1-3* |
| Gender | xx (xx.x%) | xx (xx.x%) | xx (xx.x%) | *Page 1* |
| -- Male | xx (xx.x%) | xx (xx.x%) | xx (xx.x%) | *Panel 2* |
| -- Female | | | | *Rows 1-3* |
| Ethnicity | xx (xx.x%) | xx (xx.x%) | xx (xx.x%) | *Page 1* |
| -- Hispanic or Latino | xx (xx.x%) | xx (xx.x%) | xx (xx.x%) | *Panel 3* |
| -- Not Hispanic or Latino | | | | *Rows 1-3* |
| Race | xx (xx.x%) | xx (xx.x%) | xx (xx.x%) | *Page 1* |
| -- American Indian or Alaskan Native | xx (xx.x%) | xx (xx.x%) | xx (xx.x%) | *Panel 4* |
| -- Asian | xx (xx.x%) | xx (xx.x%) | xx (xx.x%) | *Rows 1-7* |
| -- Black or African American | xx (xx.x%) | xx (xx.x%) | xx (xx.x%) | |
| -- Native Hawaiian or Other Pacific Islanders | xx (xx.x%) | xx (xx.x%) | xx (xx.x%) | |
| -- White | | | | |
| -- Other | | | | |
| Overall Interpretation of Electrocardiogram (ECG) | xx (xx.x%) | xx (xx.x%) | xx (xx.x%) | *Page 1* |
| -- Normal | xx (xx.x%) | xx (xx.x%) | xx (xx.x%) | *Panel 5* |
| -- Abnormal/Not Clinically Significant[1] | xx (xx.x%) | xx (xx.x%) | xx (xx.x%) | *Rows 1-4* |
| -- Abnormal/Clinically Significant[1] | | | | |

Because each category of analysis is either completely specified or has an “Other” option, all subjects will fit into just these breakdowns, regardless of the number of subjects in the study. Therefore this table may be considered as “well formed” or “completely specification driven”, as opposed to more data-driven reports (e.g. any Adverse Event table which is driven by the number of events occurring in each Panel=System Organ Class, or Medical History findings).

Adopting this 3-level of organization requires a small adjustment period but reaps myriad benefits:

- Code organization
- Code naming conventions
- Pre-data real table shells
- Possibility of multiple programmers collaborating on a large complex table
- Validation / Reconciliation process

Many production SAS shops have some sort of preferred naming convention. Using the Page / Panel / Row convention, this becomes easy and intuitive. Each separate panel of output can be built separately, and all of the intermediate datasets produced during the derivation (not source datasets that are used by multiple panels) can be safely erased at the end of each panel section of code. If specifications change or the validation process reveals problems, it becomes trivial to find the section of code responsible for a particular report section.

Once the [projmeta.projman] table is propagated with project-level metadata (from the Table Shells), one can easily write a short code segment that creates an actual output object (*demog.rtf*) without a single byte of data in the clinical database. The table will be unpopulated with any results, but a SAS-created version of the shell can be created very early on to QC against the statistician's mock-up version. This applies only to well-defined table types as above, not purely data driven tables like Adverse Events or Medical History.

The last two bullet points above require rather more careful consideration as they can result in non-standard coding practices.

Referring back to the discussion about (# of command files) *vs*. (# of output objects), another assumption many make is one command file = one coder (not counting the Validation coder; to be discussed next). This principal was followed here as well; like most project teams, each output object (T/L/F) is assigned to a particular team member; in our case, tracked in the [projmeta.projman] file. However one could imagine a very large very complex table with many separate sections (panels) that would take one coder too long to complete.

Consider the typical workflow in a Table program:

- Extract from data source (raw, SDTM, ADaM as appropriate) the desired data and subset for analysis
- Call appropriate SAS procedures to summarize or derive required output (e.g. Transpose, Summary, GLM, Lifetest, etc.)
- Combine with various text labels and user-define formats to produce report-ready SAS dataset (in the work library)
- Combination of macro calls and Proc Report to create publication ready final output (RTF, PDF…)

In our hypothetically enormous complex program, one might assign Coder 1 to code for Panels 1-3; Coder 2 for 4-10, and Coder 3 11-27. In this case one can produce four separate codes; #1, 2, and 3 are independent from one another, following the work flow:

- Extract from data source (raw, SDTM, ADaM as appropriate) the desired data and subset for analysis
- Call appropriate SAS procedures to summarize or derive required output (e.g. Transpose, Summary, GLM, Lifetest, etc.)
- Combine with various text labels and user-define formats to produce report-ready SAS dataset (in the work library)
- Store resulting "publication ready" series of results into a permanent SAS data set within the project library. Codes # 1, 2, 3 will produce **no** publishable output on their own.
- Codes section # 4 simply takes the data file of results as produced by executing section (1, 2, 3) and writes out the final (large) publishable report object.

The above example, while theoretically straightforward is pure conjecture and may easily have hidden issues and be impractical. However it is another potentially useful technique, and regardless leads easily to the next topic. First consider a prefix to our standard workflow, and a slight variation on the table-building process.

The principles for producing publishing-ready table (and listing) output include:

- Create a ready-to-use RTF output file (no manual post-processing required)
- Use ODS (Output Delivery System) and Proc Report to create the file (standard macro calls to automate)
- Within the RTF file, primary programmer is responsible for pagination, fully specifying all attendant column headers, labels etc. Titles and footnotes are automatically supplied by extracting from [projmeta.projman] based on program name supplied by [%getPgmName()]

- Since SAS output is directed out to the RTF destination, the usual SAS distinction of numeric vs. text variables becomes moot. All output is stored in text fields, even if the content of most cells consists only of numeric values
- Where possible, allow Word to perform formatting function; a combination of SAS styles, style overrides, and judicious use of RTF in-line codes can easily accomplish myriad tasks.
- Aside from the structural fields [`PPR`], column names are generic: e.g., the leftmost column (usually containing explanation text) is by convention `COL0`, while subsequent dose-specific columns are merely `COL1, COL2`… as specified by the Table Shells template.

Before the original source data extraction, build a "scaffold" or mold to hold the data.

```
Data final;
  format pagenum panel row 4.;
  length col0 $ 200
         col1 – col3 $ 20;
  retain pagenum 1;
  do panel = 1 to 5;
  if      (panel eq 4) then rows = 7;
  else if (panel eq 5) then rows = 4;
  else                      rows = 3;
  do panel = 1 to 5;
    do row = 1 to rows;
      output;
    end;
  end;
  drop rows;
run;
```

The code above builds a SAS data set ready to hold the contents as subsequent code section build the report. Note that fields [`pagenum panel row`] are now keys to the results. Note that within a study, most reports will have the same generic structure with respect to number and labelling of column headers. After each separate section has completed a panel of results, add them to the growing results by very simple code:

```
data final;
  update
    final
    panel1;
  by pagenum panel row;
run;
```

At the very end of the Table code, now that the [`work.final`] dataset is fully populated write to output object:

```
%rpt() /* standard macro uses output from %getPgmName() to derive output
file*/
proc report data = final nowindows…;
  cols (pagenum panel row col0 col1 col2 col3);
run;
%rpt() /* Automatically closes output object and restores LISTING dest*/
```

Since the above code only varies slightly within a particular project, it is simple in the SAS DMS environment to store a large amount of "skeleton" code to a keyboard macro, which can then be expanded and inserted into the Editor with a few keystrokes.

Lastly, consider how building tables with meta-structure fields [PPR] can aid and indeed transform the least automated, most human-intensive part of our overall workflow: Validation. Since our industry is so strictly regulated, it is insufficient to merely perform perfect work; we must also provide evidence that we have validated our work. Unfortunately this nearly doubles our workload; and still does not guarantee "perfect" results (both Primary and QC programmers are quite capable of programming different codes to the same misunderstood specifications). Typically, this workflow looks something like:

- First (Prime) coder writes code to produce a particular output object (e.g. *demog.rtf*). This SAS code completes with all results in the SAS `work` library which is naturally discarded upon program completion. The only "bread crumbs" left behind are the SASLOG, LISTING (perhaps), and whatever output object(s) created outside the SAS system.
- Secondary (QC) coder writes independent code, based on the same specifications (SAP, Tables Shells). The coder creates the same results in parallel, but typically expends no efforts to make any publishable output, instead relying on a simple Proc Print to the LISTING destination.
- QC coder now compares the results from the QC code output to the nicely formatted results from the Prime program execution. This is often accomplished using what the U.S. Military refers to as "Mark-I eyeball", meaning by pure (simple but tedious) physical inspection.

There are many ordinary circumstances where this workflow makes perfect sense and is as efficient as possible. This especially includes any circumstances when the P-QC coding takes place on an already locked database; any discrepancies between Prime and QC output can usually be quickly tracked to unexpected data, insufficiently explicit specifications, or simple programmer error (on either side). However, consider the workflow for a lengthy, Phase III registration trial planned to form the foundation of a Regulatory Submission for approval:

- Potentially long patient enrollment times and/or follow-up times
- Specifications (SAP, Table Shells) available for extended periods, although subject to change before Database Lock
- New and newly corrected data arriving throughout the trial.

Under these circumstances, the above workflow shows its weakness: Every time a single data point changes:

- Re-execute Prime program
- Re-execute QC program
- Re-apply "Mark-I eyeballs" to manually compare Prime to QC results.

As the QC process is often left to more junior staff, perhaps this inefficiency is considered acceptable. Aside from the possibly abysmal effect on the morale of the QC coder, it is also extremely easy for the eyes to overlook real discrepancies while perusing potentially thousands of pages of output. The SAS user community has produced some clever and innovative solutions over the years, resulting in many interesting papers. The solution described here takes a slightly different approach.

In studies where automated QC process is practical, the study includes an extra libref, [`QcData`]. Near the end of each analysis program, the Prime coder adds a simple data step, at the point in the flow where all results have been computed and are ready to be sent to a report object:

```
data qcdata.&pgmname;
  set final;
run;
```

Here <`&pgmname`> is a global macro symbol containing the name of the code being executed. Note that [`qcdata.&pgmname`] has the generic structure [`pagenum panel row col0 col1… colN`]. If examined by a SAS table viewer (e.g. ViewTable or FSView), the visual results would be identical to the nicely formatted publishable version stored in the RTF output file.

The QC coder now begins the QC process by creating a temporary copy of the metadata

```
data qc;
  if 0 then set qcdata.&pgmname;
  /* more code to build the scaffold for the desired table structure*/
run; /* OR */
proc sql;
  create table qc
    like qcdata.&pgmname;
quit;
```

The QC coder then proceeds to program the rest of the table according to the same specifications as used by the Prime coder. As each panel is derived and ready for comparison, it is added to the [`work.qc`] table until the process is complete. At this point, the QC coder does not [`Proc Print`] the results table, but rather calls an enterprise level comparison macro, which performs a [`Proc Compare`] of the temporary QC table to the permanent QCDATA table, using [`pagenum panel row`] as keys. The macro extracts from [`Proc Compare`] the number of differences encountered (hopefully zero!) and prints out a discrepancy report. Furthermore, it uses the program name to update [`projmeta.projman`] with the results of the comparison and the QC code execution date. This allows the Lead Programmer to scan [`projmeta.projman`] to determine which Tables have been through the QC process, how many differences were detected, and when.

This is a crude measurement, as a difference of one can be much more complex and troubling a problem than 1,000 differences detected. However, as the project work progresses, more Tables are completed and more QC checks result in the same (e.g. zero differences) results, the Project Lead can track progress and identify tables that might require more attention or resources. Other project utilities highlight which entries in [`projmeta.projman`] have no code started and the age of the various report objects.

**Next Steps**

Everything described in this paper has been tested and proved in the crucible of experience, including the Sponsor's and study teams first Application Filing. Following the spirit of *kaizen*, it is a work in progress with enormous room for growth and improvement. The strategy of having in-the-trenches coders create and polish their own tools to optimize the workflow means that progress is slow and sometimes fitful (study timelines take precedence!), but it also ensures that only useful and helpful infrastructure is developed, tested, and implemented.

**Conclusion**

The work life of the modern Clinical Statistical Programmer is quite complex; knowledge of pharmaceutical and regulatory requirements; constantly evolving and expanding CDISC standards, constantly improving software from SAS Institute, and the myriad details and responsibilities of supporting the clinical trial and safeguarding human health is daunting. Given this environment, it makes sense to build as supportive and automated work environment as possible.

**References**

Imai, Masaaki (1986). *Kaizen: The Key to Japan's Competitive Success*. New York: Random House.

Hamilton, Paul. "ODS to RTF: Tips and Tricks". http://www2.sas.com/proceedings/sugi28/024-28.pdf

Rohini Rao, Omeros Corporation. "Gilding the Lily: Boutique Programming in TAGSETS.RTF". http://www.pharmasug.org/proceedings/2012/DG/PharmaSUG-2012-DG11.pdf