PharmaSUG 2014 - Paper AD07

Techniques for managing (large!) batches of statistical output

Brian Fairfield-Carter, inVentiv Health Clinical, Cary, NC

Tracy Sherman, inVentiv Health Clinical, Cary, NC

ABSTRACT

When delivering batches of statistical output to a client, some fairly basic conditions need to be met: output must be current (generated without error from current analysis data), validation status must be current, and the collection of output must comprise what the client is expecting to receive (no missing files or extra files). While these conditions are fairly easy to enforce with small batches of output, they can be surprisingly difficult to meet with larger batches: consider for instance the practicality of checking that each of 350 entries in a programming plan has a corresponding output file (or conversely that each file has a corresponding entry). Techniques for managing a small delivery tend to be inadequate for large deliveries, and as the size and complexity of the delivery increases, the need for automation in meeting/confirming these conditions also increases.

Since large batches of output are difficult to navigate, clients will often (and sometimes without advance warning) request enhancements such as a hyperlinked Table of Contents (TOC) and/or bookmarked collated file. These can be prohibitively time-consuming to create manually, but automated methods may involve technology that is unfamiliar to most SAS® programmers.

This paper discusses problems in managing large batches of statistical output, and offers practical and automated techniques for handling the various steps in assembling a delivery, including

- Batch-updates to production and QC output
- Word/RTF document comparisons
- Checking for agreement between programming plan and output
- RTF, PDF, & Excel File collation and hyperlinked TOC generation

INTRODUCTION

With the continual push to reduce drug-development times, and the associated efforts to try and force clinical research into a manufacturing paradigm, one of the things we tend to see is that timelines for delivering statistical output (for instance, the interval between database lock and delivery of results) are less than generous. The assumption typically made is that with advance planning and robust program development, output can be refreshed and delivered with clock-like precision, at the push of a button. The reality is that even with pre-planning and (reasonably) robust code, producing and organizing a delivery tends to be a flurry of last-minute scrambling: with 'upstream' activities in Data Management also enjoying very lean timelines, the window for refreshing and re-validating analysis datasets is very narrow, but at the same time a new data cut raises the possibility of adjustments to data-handling rules, and statistical review may prompt further last-minute revisions that distract from the more rote tasks of simply producing output.

What if those conditions (pre-planning and robust code) aren't actually met? Think for a moment how long it's been since you, as a lead programmer, actually started a project 'from scratch', and had the leisure of building in at the outset all the tools and organizational niceties that give you a sense of 'keeping things on track'. If you're lucky, this hasn't become a rarity, but too often what we actually work on (especially in the CRO industry) is an endless succession of 'inherited' projects, some of which may have been well-organized from the outset, but many of which will have developed in a haphazard manner in reaction to specific, spur-of-the-moment demands, often by programmers with no sense of 'ownership' and no interest in investing in planning or organizational groundwork. Combined with this is the considerable variability in experience and proficiency across project teams: some programmers have extensive experience and have a good sense of what to look for and what to defend against, while others, quite understandably, do not (and this gap, one might venture to say, is seldom addressed by any kind of formal training).

How then does one buffer against these factors and competing demands, and remain reasonably confident that deliveries will go out on schedule and at a reasonable time of day (i.e. not late at night on the day of delivery)? While not every contingency can be anticipated, there are a number of discrete tasks that can be identified, planned for and streamlined, and there are specific points at which automation will give that life-saving boost in both speed and accuracy. Processes should strive for simplicity, and automation should take a 'minimalist' approach: consider the benefits of simple architecture that is intelligible to large numbers of programmers, as opposed to a rat's nest that only you are able to decipher. Complicated, 'do-everything' tools tend to be maintainable only by their developers, and as such usually have short life-spans, while minimalist tools that target specific tasks tend to be accessible to larger numbers of programmers, and are more likely to be user-maintainable and adaptable. Minimalist and simple tools can also be 'retro-fitted' more easily to existing projects, which is of tremendous benefit to 'inherited' projects that suffer from a lack of early planning, from multiple changes to project teams, and from the various forms of neglect and divergence that can afflict projects in this age of 'dynamic' project teams.

The first step in setting up for a delivery is to clearly understand what the expectations and tasks are, so that you can plan accordingly. If we take database lock as a common reference point, we know that we'll get an updated data snapshot, meaning we'll have to refresh and re-validate analysis datasets, followed by the same for Tables/Figures/Listings (TFLs). Its often blindly assumed that re-running TFL programs will produce everything required for the delivery, and the task of confirming that all required files have *actually* been generated is often overlooked, but in reality there are a variety of reasons a file might fail to refresh, one being that the file is locked simply because it has been opened by another user. And depending on specific client requirements, there may be other components to the delivery that need to be created. So we start by listing the tasks that need to be performed:

- Batch-update to datasets and production output (with comprehensive log review), and batch-update to (and review of) validation output
- Confirm a 1:1 match between programming plan and output (all required files are created, and no stray/stale/artifact files exist)
- (Discretionary) compare updated production output against prior/archived output and confirm that no unexpected changes have taken place
- Other requirements: File Collation and Table of Contents creation, Excel workbook creation

For a delivery that consists of a handful of output files, it might be feasible to perform all of these steps manually: run each program individually, scan through the log files, manually compare QC output against production output, cross-check output files against the programming plan, and even assemble output files into a single collated document by hand. For larger deliveries, timelines and the need for accuracy would both prohibit this approach: it's not uncommon for QC and statistical review to continue until late on the day of delivery, meaning that meeting the delivery deadline while employing a 'brute force' approach would be physically impossible. Consider for example that the task of creating a collated file with hyperlinked Table of Contents for 200 output files would likely take in excess of 5 hours, and is therefore not something that any sane person would embark on at 6:00 pm on the day of a delivery. Worse however, is the potential loss of accuracy that a manual approach would produce when applied to a large delivery: manually checking QC output against production output is notoriously error-prone, as is the process of checking output filenames against entries in a programming plan. Anything that is sensitive to programmer fatigue, when combined with that rushed, end-of-the day sensation, is a recipe for disaster.

Additionally, there is something slightly mysterious about the properties of a large batch of output, as though the increase in file count has a multiplicative rather than additive effect on the effort required. This might seem counter-intuitive, but consider the effect on your role as 'project lead' of adding just a couple programmers to the team: the possibility for conflicting interpretations and inconsistency can grow drastically, as can the potential for bottlenecks and unpredictability in the ebb and flow of work completion. Ironically, the restaurant industry probably has a better grasp of this phenomenon than we do, since most restaurant menus include a statement along the lines of

A 15% gratuity will be charged for parties of 10 or more.

In other words, there's some recognition that the effort expended does not increase in a linear fashion with increasing demands and complexity, but probably shows a much less predictable relationship:

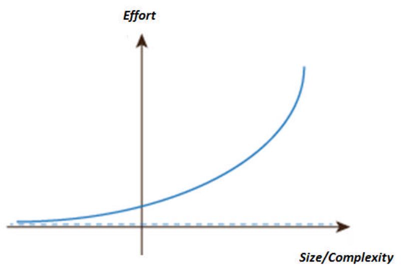


Figure 1. With increasing size and complexity, the effort in preparing a delivery seems to increase in something other than a simple linear fashion.

The net result is that with large deliveries, the effort we apply and the techniques we use can't simply form a larger version of a small delivery; we can't get away with just 'doing more' of what we'd do for a small delivery, since although the components are all readily recognizable, the interactions result in a process that is stochastic.

But like the gratuity a restaurant charges, we do have at our disposal various ways of compensating: we can anticipate and plan around the problems we're likely to encounter, and we can make use of well-placed automation to provide enhancements and to perform specialized checks that fall outside typical QC practices (but which are nonetheless vital to ensuring timeliness and accuracy in a deliverable). The following sections describe the major steps in preparing a delivery, and offer recommendations and technical solutions for making the process faster and more accurate.

BUT FIRST, A NOTE ON WINDOWS SCRIPTING…

In the tightly-controlled environment that we as statistical programmers work in, there tends to be very little scope for working in anything other than Base SAS: licensing other SAS products is expensive and difficult to justify, and IT departments tend to take a dim view of any attempts to install non-standard/non-approved software. At the same time, various tasks discussed in this paper require the ability to work with file systems, and to 'drive' applications such as Microsoft Word, neither of which are particularly convenient in Base SAS (aside from the clunky and long-deprecated 'Dynamic Data Exchange' (DDE), there really isn't any means of inter-process communication in Base SAS that would enable us to 'drive' MS Word).

The challenge then is to make the best use of what we have available: certain file-system tasks can be performed using DOS commands and batch files, and automation in Microsoft applications such as Word and Excel can be achieved using Visual Basic and built-in macro-generating facilities. There is however a very simple scripting interface built into the Windows operating system, which arguably offers the greatest simplicity and versatility when it comes to handling these file system and applications-interaction tasks: the Windows Scripting Host (WSH). For additional detail, refer to Hunt et al, 2004/2005, but in short, what the WSH offers is the ability to write 'bare-bones' Visual Basic (the 'Vbscript flavor', if you like), save it to a text file with '.vbs' file extension, and simply double-click the file to execute (or alternately, execute the script via an 'x dos' command from SAS if you happen to be working in Windows SAS).

Vbscript is probably the easiest 'extra-curricular' programming language for a die-hard SAS programmer to learn, in no small part because so often it can be reverse-engineered from the Visual Basic code created by the macro-generating facilities built into Microsoft applications. For instance, say you wanted to figure out what the Vbscript code to save a Word document as an RTF file would look like; you could start by opening the file in Word, and recording a macro while doing a manual 'save as' to RTF. The generated Word Visual Basic code would look something like this:

```
ActiveDocument.SaveAs FileName:="temp.rtf", FileFormat:=wdFormatRTF, _
    LockComments:=False, Password:="", AddToRecentFiles:=True, WritePassword _
    :="", ReadOnlyRecommended:=False, EmbedTrueTypeFonts:=False, _
    SaveNativePictureFormat:=False, SaveFormsData:=False, SaveAsAOCELetter:= _
    False
```

This might look slightly daunting, and certainly if you just dropped it verbatim into a Vbscript, the script wouldn't run, but there are really just two things to take note of:

1. The macro, since it is internal to Word, doesn't need an 'object reference' to the instance of Word.
2. Many of the arguments to the 'SaveAs' function consist of internal Word constants, and many of them can be ignored altogether

So what does this mean? Let's see what the equivalent Vbscript would look like, and it'll become clear:

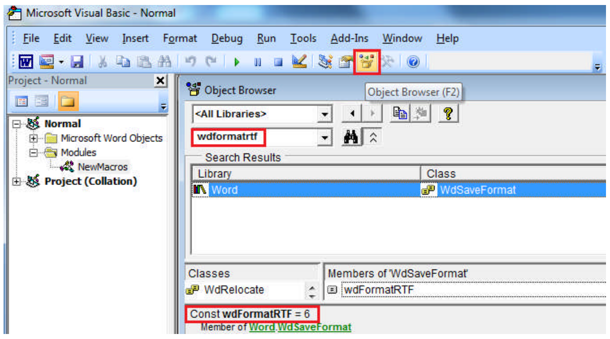
```
Set wd=CreateObject("word.application")
…
wd.ActiveDocument.SaveAs "temp.rtf", 6
```

In other words, before the Vbscript can call Word functions, it needs to launch Word (via 'createobject'), and the functions have to reference the instance of Word (hence '***wd***.ActiveDocument.SaveAs'). Further, the RTF file type is specified as file format '6', where referencing this constant using the 'wdFormatRTF' variable would otherwise be meaningless to the Windows Scripting Host. And how would one know that wdFormatRTF=6? Fortunately these values can be looked up in the 'object browser' in Word, from the 'Developer' tab (Figure 2, below). Note that if you don't see the Developer tab in Word you can add it in by following these steps:

1. Click the Microsoft Office Button, and then click Word Options.

2. Click 'Popular', and then select the 'Show Developer tab' in the Ribbon check box.

3. Once you can see the Developer tab, click Visual Basic and then Object Browser (or F2).



```
Const wdFormatRTF = 6
Member of Word.WdSaveFormat
```

Figure 2. Function syntax and constants can be looked up on the 'Object Browser' in Word (here 'wdformatrtf' has been entered in the 'search' box, with the value of 6 displayed in the results box).

Finally, of all the arguments specified in the Visual Basic fragment, only the file name and file type are actually necessary in order to perform the task of saving the file as RTF. Ultimately, these features make it very easy to get started with Vbscript, and allow you to 'learn as you go'.

BATCH-UPDATES

Although batch-updates just involve 're-running everything', there are a number of pitfalls to beware of. First, the process needs to recognize dependencies: analysis datasets need to be refreshed before the TFLs that depend on them, and among analysis datasets there is usually some form of hierarchy (for instance, analysis datasets will usually include variables from a subject-level dataset such as ADSL, and this subject-level dataset must be refreshed prior to any other datasets that depend on it).

Second, although it may seem superficially attractive, a batch-update process that utilizes 'interactive mode' should really be avoided. Interactive mode introduces the possibility of hidden dependencies, cross-contamination and unintended artifacts: global macro variables initialized by unrelated programs, file references and output destinations (ODS or printto) left open, and discrepancies between autocall macros and '%included' macro definitions can all lead to unintended (and in some cases disastrous) results. These issues represent a gamble that should never be taken when preparing a delivery to a client. Interactive mode is useful during code development (arguably the only purpose for which it is intended). Although it is possible to establish programming conventions that if enforced would make interactive batch-updates fairly safe, the best way to avoid these hidden dependencies and artifacts is to simply require that programs all be written so that they can be submitted independently in batch mode. This means that each program should '%include' any required setup programs (or alternately use a default 'autoexec' file), and where applicable should call any post-processing macros required for creating production output (i.e. for creating word-processor-ready RTF files), as opposed to relying on some sort of 'wrapper' supplied by the batch-update process (otherwise this impedes the write-run-check cycle that should define program development).

Explicit log-file redirection should also be avoided. Although there are a few alternate ways of creating log files (manually saving from the log window or using 'dm save' to capture the contents of the log window and save to file, or using 'proc printto' log-file redirection, even cluttering programs with code that detects submission environment (batch versus interactive) and handles log output accordingly), by far the most effective and intuitive, and the least error-prone means of handling log output is simply to apply this rule:

> ***Review log output in the log window during interactive code development, and otherwise use the default offered by batch mode***.

For programs submitted in batch mode, a log file is created in the same directory as the program, having the same filename root (but with 'log' file extension) as the program, and this is without question the ideal when it comes to managing log output. Since a log file reports on the functioning of the *program*, it makes little sense to have it named with a root other than that of the program. Yet in many cases log files generated via proc printto are assigned names like "table_123.log". Also, again given the divergence that can invade project teams, inconsistent implementation of explicit log-file redirection can result in cross-contamination of log messages (printto destinations left open in interactive mode, such that the log file receives log output from more than one program), log files being written to different directories (particularly a problem when projects require separate blinded and unblinded directories), and incomplete or fragmented log files (i.e. when a log printto destination is opened at the top of the program, and then inadvertently closed inside a post-processing macro rather than at the end of the program). All of these things can make it virtually impossible to either confirm correct functioning, or diagnose incorrect functioning.

The first line of defense in any batch-update process should be good house-keeping, or in other words cleaning up unnecessary files. Retired programs should be deleted, or at least moved to an archive directory, as should log files. This avoids accidentally running decommissioned programs, or reviewing old log files.

Output files (i.e. RTF files) should also be cleaned out prior to a batch-update, but it is often useful to archive these so that a directory-wide comparison between prior and current production output can be run, which often helps in checking the results of incremental changes to source data and in focusing QC efforts (this will be discussed in greater detail in a later section).

Batch-update methods

There are many of ways of setting up a batch-update, but especially in this age of rapidly-changing project teams, simplicity is invariably preferable, for example:

```
x "cd <directory>";
x 'SAS -SYSIN t_dmg.sas';
x 'SAS -SYSIN t_ae.sas';
x 'SAS -SYSIN t_ecg.sas';
(etc.)
```

This will, from either an interactive or batch SAS session, run each specified program in batch-mode, in an independent SAS session, producing a log file in the same directory as the program file and sharing the same filename root. For derived datasets extra care must be taken to ensure that programs are listed in order of dependency (i.e. so that a subject-level dataset like ADSL is created before any datasets that depend on it). While there are more sophisticated methods for handling these dependencies (i.e. the 'make' facility (see Fairfield-Carter and Hunt, 2008)), in the interest of simplicity it is usually preferable to just use program-naming conventions that make the hierarchy obvious: d0_adsl, d1_adae, d2_adef, etc., and to construct the batch-update list accordingly.

It is also possible to generate the program list dynamically (i.e. from the programming plan, or by 'piping' a directory listing and selecting files with '.sas' file extension, comparable to the log-check macro shown below), but the benefits of doing so must be weighed against the costs of additional complexity; a dynamically-generated list does not in of itself guarantee that every required output gets generated, so it may be preferable to keep the batch-update as simple as possible, and invest instead in checking output against the programming plan (note also that dynamically-generated lists will be more safely applied to TFLs, where there aren't internal dependencies, than to analysis datasets). The bottom line is that in even relatively large and complicated projects, programs will likely number in the tens rather than hundreds, particularly if analysis datasets have been efficiently designed and if TFLs sharing structural similarity are generated by single rather than by multiple programs. This should be weighed when considering how complicated a batch-update system should be permitted to become.

Directory-wide log-checking

Most programmers either have or have access to macros that perform directory-wide log-checks (typically something that pipes the output from an 'x dir' command in order to generate a list of log files in a target directory, which is then iterated through so that each log file can be opened and scanned for disallowed log messages), but for the sake of completeness a simple version is presented here.

Point to the target directory, and generate a list (and a count) of log filenames

```
x "cd <target directory>"; *** (i.e. F:\projects\ABC\programs);
filename test_ pipe 'dir /B';

data filelist;
  attrib str_ length=$200.;
  infile test_ LRECL=200 truncover;
  input str_ $200.;
  if index(upcase(str_),".LOG") then output;
run;

proc sql noprint;
  select count(*) into :nfiles from filelist
  ;
quit;
```

Set up for log file and log-summary file handling

```
%let filrf=mylog; %*** FILE REFERENCE FOR EACH LOG FILE;
```

```
      %let sumrf=mysum; %*** FILE REFERENCE FOR LOG-SUMMARY FILE _log_summary.txt;

      %let rc=%sysfunc(filename(sumrf,"_log_summary.txt"));
      %let fid_=%sysfunc(fopen(&sumrf,w)); %*** OPEN LOG-SUMMARY FILE FOR WRITING;
```

Iterate through the list of log files, and open each log file in turn

```
      %do i=1 %to &nfiles; %*** FOR EACH LOG FILE IN THE CURRENT DIRECTORY;

        data _null_;
          set filelist;
          if _n_=&i then call symput("logfile",trim(left(str_)));
        run;

        %let rc=%sysfunc(filename(filrf,"&logfile"));
        %let fid=%sysfunc(fopen(&filrf)); %*** OPEN LOG FILE FOR READING;

        %let rc=%sysfunc(fput(&fid_,LOG FILE &i:: &logfile));%***WRITE OUT FILE NAME;
        %let rc=%sysfunc(fwrite(&fid_));
```

Read each log line

```
        %do %while(%qsysfunc(fread(&fid))=0);
          %let rc=%qsysfunc(fget(&fid,c,200)); %*** CAPTURE EACH LOG LINE;
          %let c_=%superq(c);
```

Test for disallowed log messages

```
          %if  %index(%qupcase(&c_),ERROR)>0
            or %index(%qupcase(&c_),WARNING)>0
            or %index(%qupcase(&c_),UNINI)>0
            or %index(%qupcase(&c_),INVALID)>0
            or %index(%qupcase(&c_),MERGE STATEMENT)>0
            or %index(%qupcase(&c_),MATHEMATICAL OPERATIONS COULD NOT)>0
            or %index(%qupcase(&c_),OUTSIDE THE AXIS RANGE)>0
            or %index(%qupcase(&c_),W.D FORMAT)>0
            %then %do; %*** WRITE ONLY DISALLOWED STATEMENTS TO LOG SUMMARY;
```

Write out any disallowed messages to the log-summary file

```
            %let rc=%sysfunc(fput(&fid_,&c_));
            %let rc=%sysfunc(fwrite(&fid_));

          %end;
        %end;
```

Clean-up

```
        %let rc=%sysfunc(fclose(&fid)); %*** CLOSE LOG FILE;
        %let rc=%sysfunc(filename(filrf));

      %end;

      %let rc=%sysfunc(fclose(&fid_)); %*** CLOSE LOG-SUMMARY FILE;
      %let rc=%sysfunc(filename(sumrf));
```

As mentioned previously, cleaning out and/or archiving log and output files prior to a batch-update is always a good idea, but failing that, after a batch-update, all output files should have the same file system date (the log check will indicate where any programs failed, but a quick look at file system dates will also provide immediate insight). Be particularly careful to check for any analysis datasets that fail to update; too often this happens because someone has opened the dataset in the SAS Viewer, which locks write-access to all other users (the SAS Viewer should really be classified as 'malicious software' because of this propensity for interfering with dataset updates, and its use in opening datasets on shared directories should really be strictly prohibited).

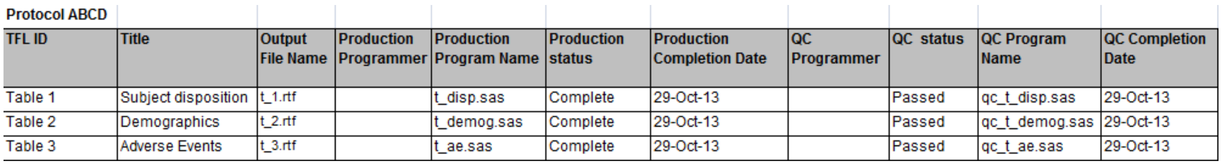
Once the batch-update has been completed on the production side, and the directory-wide log check shows that everything ran cleanly, a similar batch-update is done for QC output. As with production programs, QC programs should be written such that they can be run independently in batch-mode, and should be designed to produce clean output when there is a match between production and QC. In other words, once code development is complete on the QC side, there should be no stray 'proc print's or other output, but simply a 'proc compare' listing that can demonstrate at a glance a match between production and QC. For large deliveries it is absolutely not feasible to support cluttered QC output (i.e. where you have to wade through a bunch of intermediate listings before finding the crucial 'proc compare' output) and it is especially not feasible to have QC output that requires the manual comparison of QC values to production values. Ideally, the confirmation of clean QC output would be summarized automatically, by the combination of a directory-wide log check confirming clean log files and a macro that parses QC output and identifies any discrepancies between production and QC.

Once the validation cycle is completed, which essentially confirms that the contents of each output file agrees with its associated specifications, the next step is to confirm that all of the files specified in the programming plan were in fact generated, and that no extraneous files were created.

CONFIRMING A 1:1 MATCH BETWEEN PROGRAMMING PLAN AND OUTPUT

In a large project, it is quite possible for 'retired' programs to be accidentally run, creating output that is no longer required for the project, and by the same token it is possible that a required program will be missing from the batch-update process, meaning required output isn't generated. It's therefore essential before going ahead with a delivery to confirm not only that the content of individual files is correct, but that the suite of files itself is correct.

Projects will typically have some sort of 'programming plan', often in the form of an Excel spreadsheet, that lists the required output for each delivery: output file name, table/figure/listing identifier, title, program that generates the output, program author, program status, QC status, etc.:



Protocol ABCD

TFL ID	Title	Output File Name	Production Programmer	Production Program Name	Production status	Production Completion Date	QC Programmer	QC status	QC Program Name	QC Completion Date
Table 1	Subject disposition	t_1.rtf		t_disp.sas	Complete	29-Oct-13		Passed	qc_t_disp.sas	29-Oct-13
Table 2	Demographics	t_2.rtf		t_demog.sas	Complete	29-Oct-13		Passed	qc_t_demog.sas	29-Oct-13
Table 3	Adverse Events	t_3.rtf		t_ae.sas	Complete	29-Oct-13		Passed	qc_t_ae.sas	29-Oct-13

Figure 3. Example programming plan.

With a small number of output files it's fairly easy to confirm that each file listed under the 'Output File Name' column is actually created through the batch-update process (and that there are no additional output files beyond what's listed in the programming plan), but when the output files number in the hundreds this becomes both time-consuming and difficult to do accurately (particularly a problem given that batch-updates often have to be repeated several times in the days and hours immediately prior to a delivery).

It is however, very easy to pull a list of required output files from the programming plan, then generate a list of RTF files in the directory that production output is written to, and compare the two lists. While there are various ways of reading Excel data directly (i.e. ActiveX Data Object (ADODB) scripting), for the sake of simplicity what we opted for here is a workable but non-sophisticated method: simply dropping the first three columns of the programming plan into a comma-delimited text file which is then read (by a script, but this could equally easily be done with a SAS macro, with a directory listing generated using methods shown in the log-check macro above) which checks each entry against the list of output files sitting in a target directory.

The following information is extracted from the programming plan into a comma-delimited text file (note that the check only really requires the file name (the third element), but that the other two elements (TFL identifier and title) can be useful for diagnostic purposes):

```
Table 1,Subject disposition,t_1.rtf
Table 2,Demographics,t_2.rtf
Table 3,Adverse Events,t_3.rtf
```

The contents of the text file are loaded into a 2-dimensional array, and the list of RTF files in the target directory is loaded into a second array. Each array is then compared to the other in order to identify any entries in the programming plan that don't have corresponding RTF files, and any RTF files that don't have corresponding entries in the programming plan (missing RTF files may signal programs that failed or were not included in the batch-update process, and extra RTF files may indicate that 'retired' programs were accidentally run, or that the programming plan needs to be updated, or that programs contain incorrect filename references).

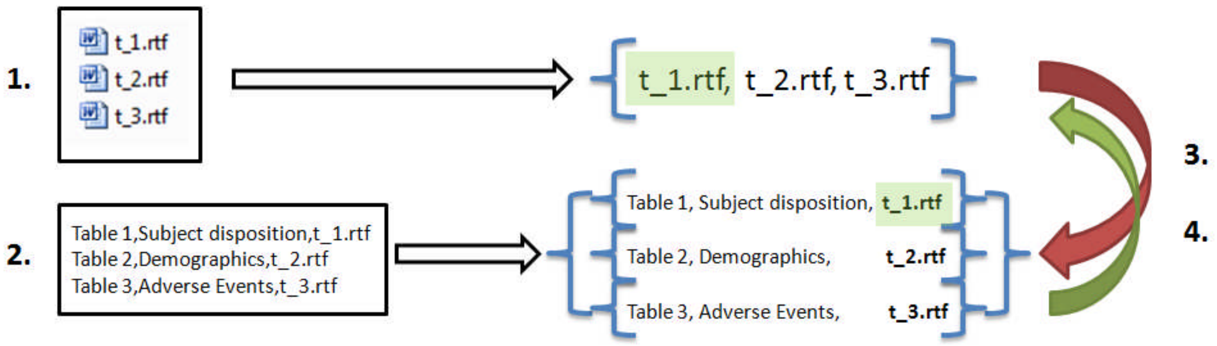

1.
t_1.rtf
t_2.rtf
t_3.rtf
t_1.rtf, t_2.rtf, t_3.rtf
2.
Table 1,Subject disposition,t_1.rtf
Table 2,Demographics,t_2.rtf
Table 3,Adverse Events,t_3.rtf
Table 1, Subject disposition, t_1.rtf
Table 2, Demographics, t_2.rtf
Table 3, Adverse Events, t_3.rtf
3.
4.

Figure 4. Steps in checking for a 1:1 match between programming plan and output:
1. **Write the RTF file list to an array**
2. **Write the TFL ID, Title and output filename (from the programming plan) to a 2-dimensional array (only the filename element is strictly necessary)**
3. **Iterate through the file array from step 1 and check for missing programming plan entries**
4. **Iterate through the filename array from step 2 and check for missing RTF output files**

The script creates a text file which is empty if there is a perfect 1:1 match between the programming plan and the collection of output files, and otherwise lists out discrepancies:

```
1. On programming plan but not among RTF output: t_2.rtf  |  Demographics
2. Among RTF output but not on Assignments file: t_9.rtf
```

In other words, the script lists two types of discrepancies: first, where output listed in the programming plan is missing from the batch of RTF output, and secondly, where extra RTF output files are created that are not included in the programming plan; this provides a convenient guide for any updates that need to be made.

Note that although the information is not strictly necessary, by including TFL title on the text file that serves as input to the script, we're able to provide this on the discrepancy report, which is useful if the file-naming conventions for the given project result in non-informative filenames for RTF output.

The complete script is provided with the demo described in Appendix 1, but the major steps are highlighted here:

1. Write RTF file list to an array

```
For each file in f.files            'Iterate through files in the directory
  filename=file.name                       'Capture the file name
  if UCase(right(filename,4))=".RTF" then  'Select RTF filenames
    file_array(file_count)=file.name       'Write filename to array element
    file_count=file_count+1
  end if
next
```

2. Write TFL ID, Title and output filename (from the programming plan, via text file) to an array

```
Set intext=fso.OpenTextFile("TFLs.txt",1) 'Open text file
While Not intext.AtEndOfStream
text_=intext.ReadLine & NewLine           'Read each line
file_array_frmfile(line_count,1)=split(text_,",")(0) 'TFL ID from prog. plan
file_array_frmfile(line_count,2)=split(text_,",")(1) 'Title from prog. plan
file_array_frmfile(line_count,0)=split(text_,",")(2) & ".rtf" 'File name
line_count=line_count+1
```

3. Iterate through the RTF output and check for missing programming plan entries

```
for i=0 to file_count-1   'Counter for output (RTF) files
  filefound=0
  for j=0 to line_count-1 'Counter for entries from programming plan
    if file_array(i)=file_array_frmfile(j,0) then
      filefound=1         'RTF file has a matching programming plan entry
    end if
  next
  if filefound=0 then 'If RTF file has not corresponding to Prog. plan entry,
                      'write discrepancy to the log
   log_.writeline "Among RTF output but not on Programming Plan:" (--etc.--)
```

4. Iterate through the programming plan entries, and check for missing RTF output files

```
  for i=0 to line_count-1     'Counter for entries from programming plan
    filefound=0
    for j=0 to file_count-1   'Counter for output (RTF) files
      if file_array_frmfile(i,0)=file_array(j) then
        filefound=1           'Programming plan entry has a matching RTF file
      end if
    next
    if filefound=0 then   'If no RTF file corresponding to Prog. plan entry,
                          'write discrepancy to the log
  log_.writeline "On Programming Plan but not among RTF output:" (--etc.--)
```

COMPARING PRIOR AND CURRENT PRODUCTION OUTPUT

Before running a batch-update to production output, it's always worth-while archiving the current set of output files: you never know when you might need to dissect changes that have taken place between versions. There are a number of instances where the ability to do a complete comparison between current and prior output is valuable, and two key ones are described here.

First, validation and statistical review activities in the days prior to the delivery tend to result in a batch of output showing a variety of different run dates, which (in order to get consistent run dates) demands a last-minute batch-update once validation and review are complete but usually at a point where there isn't enough time to do another complete check of QC output. Rather than attempting to re-check all QC output, output files can simply be archived then refreshed via batch-update, and the two sets of output compared using a Word-compare (note that while the method discussed here assumes that output files are Word documents or RTF files, the technique should also work with simple ASCII output).

Second, when a database undergoes unlock and re-lock, the net effect on statistical output tends to be fairly localized (i.e. database updates are usually limited in scope). This means that when statistical output is refreshed based on the re-locked database, the majority of output files will be effectively unchanged, and the review of QC output for those files will be redundant. Again, a comparison of pre- and post-re-lock production output can replace a complete QC cycle; specific output files where changes have taken place can be identified, and QC activities focused on those.

Implementation described here uses a script which harnesses the "ActiveDocument.Compare" function of MS Word, but instead of displaying differences in a merged document, simply writes out a difference summary to a text file:

```
COMPARISON SUMMARY FOR: <directory>\_1.rtf; NUMBER OF DIFFERENCES: 0
--------------------------------------------------------------------

COMPARISON SUMMARY FOR: <directory>\_2.rtf; NUMBER OF DIFFERENCES: 2
--------------------------------------------------------------------
w
x
```

Differences identified using this method aren't particularly informative, but the point is simply to identify those files where differences exist, and do a more detailed review of QC output for those files; the method therefore is only really applicable to instances where you are either expecting no differences, or aren't expecting large numbers of differences.

Note that if the output files include run date (for instance, if each table contains a footnote that shows something like "**Source code: <directory>\program.sas   Run date: 28Feb2014**"), then large numbers of spurious discrepancies will be listed, since run dates on current and prior output will differ. These however can be fairly easily omitted (as illustrated in step 5, below), since the placement of run date tends to follow fairly rigid conventions.

The script uses the 'currentdirectory' property, so is intended to be placed in a directory having the current output in a sub-directory (arbitrarily named 'CURRENT'), and prior output in a second sub-directory (arbitrarily named 'PRIOR').

The complete script is provided with the demo described in Appendix 2, but with major steps highlighted here:

1. Create references to directories (relative to 'currentdirectory') containing current and prior RTF output

```
set shell=createobject("WScript.shell")
set fso=createobject("scripting.filesystemobject")
CURR_dir=shell.currentdirectory & "\CURRENT"
PRIOR_dir=shell.currentdirectory & "\PRIOR"
set d=fso.getfolder(CURR_dir)
```

2. Launch Word

```
set wd=createobject("Word.application")
```

3. Iterate through current output files (filtered by RTF file type), and check for a corresponding file among the prior output

```
for each file in d.files
 if fso.GetExtensionName(file) = "rtf" then
  if fso.fileexists(PRIOR_dir & "\" & file.name) then
```

4. Open the current output file and compare with the prior file sharing the same file name

```
wd.Documents.Open(CURR_dir & "\" & file.name)
wd.ActiveDocument.Compare PRIOR_dir & "\" & file.name
```

5. Write out differences between the two files, but ignoring any that relate to run date (in this case, any discrepancies arising from the 'Source code: ' footnote)

```
For Each diff In wd.ActiveDocument.Revisions
  Set myRange=diff.Range
  myRange.SetRange myRange.Start-121, myRange.End
  if instr(myRange,"Source code: ") then
    ' IGNORE DIFFERENCES IN RUN DATE!
  else
    log_.writeline diff.range
   end if
Next
```

6. Finally, write out information on any files in the collection of current output that don't exist among the prior output (note that by reversing the arrangement of directories, the same test can indicate files among the prior output that don't exist among the current output).

```
if fso.fileexists(PRIOR_dir & "\" & file.name) then
  ...
else
  log_.writeline "***FILE " & file.name & " DOES NOT EXIST IN THE PRIOR OUTPUT"
```

ENHANCEMENTS FOR NAVIGATING OUTPUT: FILE COLLATION AND HYPERLINKED TOC GENERATION

Depending on output file-naming conventions, it may be possible to discern something of the content of a file based on file name (for instance, where an output file name includes the name of the generating program, which has in turn been given an informative name: 't_123_t_ae.rtf' probably implies a summary of adverse events). It may be though that output file names give no indication at all of content (for instance, 't_123.rtf' could contain practically anything). In any case, having some means of quickly identifying and displaying specific content is of considerable value, and is often served by means of a hyperlinked Table of Contents:

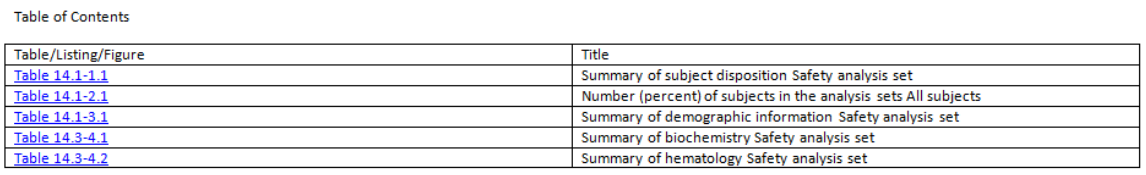


Table of Contents

Table/Listing/Figure	Title
Table 14.1-1.1	Summary of subject disposition Safety analysis set
Table 14.1-2.1	Number (percent) of subjects in the analysis sets All subjects
Table 14.1-3.1	Summary of demographic information Safety analysis set
Table 14.3-4.1	Summary of biochemistry Safety analysis set
Table 14.3-4.2	Summary of hematology Safety analysis set

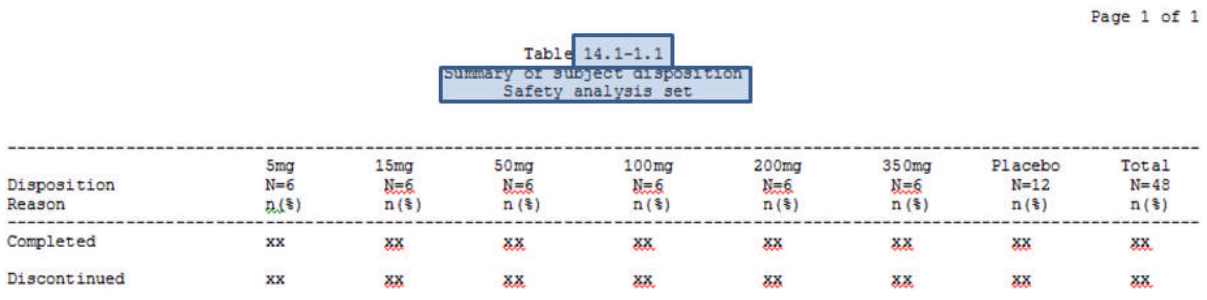
Figure 5. Even a very simple hyperlinked TOC is a valuable aid in locating and displaying specific output.

In this sample TOC, only the table/figure/listing (TFL) identifier and the title are displayed, and the TFL identifier is hyperlinked to the associated output file (output file type is basically arbitrary, but in this case is RTF). Though fairly unsophisticated, this TOC nonetheless makes it possible to quickly identify and open specific output.

In some cases clients will require that output-navigation be taken a step further: instead of simply requiring a TOC hyperlinked to the individual output files, the output files themselves are concatenated below the TOC into a collated file. The TFL identifiers are then not hyperlinked to external files, but rather to corresponding bookmarks in the collated file. The steps in assembling a TOC and/or collated file can certainly be carried out manually, but as with other techniques discussed in this paper, approaches that work with small deliveries tend to be inadequate when the number of output files increases; suffice it to say that for a delivery of more than a couple dozen tables, it is completely impractical, and probably impossible within the confines of typical timelines, to carry out by hand.

CREATING A HYPERLINKED TOC

The first step in generating a TOC is to figure out how to assemble the 'raw ingredients', those being the list of files, the associated TFL identifiers, and titles. While it may be possible to retrieve these from a programming plan or other static file, this might pose a weak point in the process since it would be sensitive to the accuracy with which information has been manually transcribed. Instead, look at where this information resides naturally: titles and TFL identifiers exist in predictable locations within output files, and file names can be retrieved in a number of ways from directory listings.



```
                                                                                  Page 1 of 1

                                            Table 14.1-1.1
                                     Summary of subject disposition
                                          Safety analysis set

-----------------------------------------------------------------------------------------------
                        5mg      15mg      50mg      100mg      200mg      350mg    Placebo    Total
Disposition             N=6      N=6       N=6       N=6        N=6        N=6      N=12       N=48
Reason                  n(%)     n(%)      n(%)      n(%)       n(%)       n(%)     n(%)       n(%)
-----------------------------------------------------------------------------------------------
Completed               xx       xx        xx        xx         xx         xx       xx         xx

Discontinued            xx       xx        xx        xx         xx         xx       xx         xx
```

Figure 6. Typical placement of TFL identifier and titles.

Output files have to follow rigid style guidelines, and predictability in the placement of TFL identifier and title lines makes the retrieval of this information amenable to automation. Where output is generated using SAS ODS (for instance, ODS RTF output destinations), titles and TFL identifiers can also be stored in and retrieved from extended file properties:

```
ODS RTF file="my_table.rtf"
title="Table 14.1-1.1 Summary of Subject disposition Safety analysis set";
```

As we've seen in previous examples, creating and working with a list of directory contents can be as easy as

```
set f=fso.getfolder(shell.currentdirectory)
for each file in f.files
```

With these two things - the predictable location of titles and identifiers and the ease of iterating through a file list - we can retro-fit or superimpose a system for generating a hyperlinked TOC on a directory containing output files, without hard-coding or having to manage any of the content by hand. Pared down to its essentials, the process requires the following:

1. Open the TOC in Word as a blank document
2. Iterate through the file list taken from the target directory
3. For each file of the specified type (i.e. RTF)
   a) Capture the file name
   b) Open, and select & copy the TFL identifier and title lines
   c) Write the TFL identifier and title(s) as a new line in the TOC file
   d) Select the TFL identifier and create a hyperlink to the file/file name captured in step 3.a

The full script is provided with the demo described in Appendix 3, but the major steps are illustrated here:

1. Open the TOC as a blank document

```
Dim wd
Set wd=createobject("word.application")
wd.documents.add "Normal", False, 0
wd.activedocument.saveas f & "\" & "TOC.doc"
wd.documents.open f & "\TOC.doc" ,,,,,,,,,0
```

2. Iterate through the file list

```
for each file in f.files
  filename=file.name
  if UCase(right(filename,4))=".RTF" then
```

3b. Select and copy TFL identifier and title lines

```
wd.Selection.MoveDown ,,1
Identifier=wd.Selection
wd.Selection.MoveDown ,,1
Title1=wd.Selection
```

3c. Write the TFL identifier and title to the TOC

```
wd.Selection.InsertRowsBelow 1
wd.Selection.typetext Trim(file_array(j,2)) 'TFL identifier
wd.Selection.MoveRight
wd.Selection.typetext Trim(file_array(j,3)) 'Title(s)
```

3d. Select the TFL identifier in the TOC, and create a hyperlink to the associated output file

```
wd.Selection.Find.execute file_array(j,2), , , , , , 1, 1
wd.ActiveDocument.Hyperlinks.Add wd.Selection.Range, file_array(j,0)
```

Practically speaking, things can become complicated in a couple of ways: first, since the file list is by default ordered alphabetically by file name, and this in turn dictates the order of entries in the TOC, re-ordering (i.e. according to TFL identifier) may be required (consider for instance the difference in alphabetical ordering that would result from these alternate file-naming conventions: "t_labs(Table 3).rtf" versus "t_3_t_labs.rtf"). Second, unless figures have been loaded into 'container' documents (where titles and footnotes are provided as simple text as opposed to being included in the graphics output, and where the graphics output is inserted between the titles and footnotes), it will not be possible to retrieve TFL identifiers and titles using the same methods that work for tables and listings (the same may also be true of tables created as ODS output, depending on whether or not titles are set as part of the document body).

The first issue is relatively straight-forward to handle, provided the required ordering follows logically from the TFL identifier; in some cases the TFL identifier is integral to (and can be parsed out of) the file name (for instance, if file names follow a convention like "t_ae_14_2_3.rtf"), or failing that, can be pulled out of the title. The relationship between file name and TFL identifier can then be used to re-order the file list before building the TOC.

Handling the second issue will depend on how the reporting environment is set up, but a number of options exist (a bit of creativity will be required in tailoring the solution to the particular instance). First, as mentioned previously, ODS allows you to set certain extended file properties, including 'title', which can then be retrieved in Vbscript via

```
title_=wd.ActiveDocument.BuiltInDocumentProperties("Title")
```

In this case, the file has been opened in Word, making the title 'visible' in the active document. Alternately, RTF files can be read as simple text, and the title property parsed out:

```
text_=intext.ReadLine & NewLine
if instr(text_,"{\title ") then
(etc.)
```

If all else fails (i.e. ODS file properties can't be used, and conventions governing title placement haven't been followed consistently), and hard-coding titles into the TOC is starting to seem like an option, consider 'bootstrapping' with an auxiliary file as an alternative: since titles (and footnotes) are available in SAS metadata tables, they can be captured while each TFL program is running:

```
proc sql;
    create table titles as select * from sashelp.vtitle where type="T";
quit;
```

The TFL identifier can then be parsed out of the collection of title lines, and with the associated titles written out to a text file. For instance, in a figure-generating program we'd somewhere have 'title' statements (either hard-coded or generated dynamically), assuming we're not creating titles via Annotate statements:

```
title1 "Figure 16.2.5-1.2b Individual plasma concentration-time profiles (Per
protocol)";
```

Right after the title statements, you could imagine calling something like

```
%mtitlefile; *** (APPEND TITLES AND TFL ID TO AUXILIARY FILE);
```

, which would capture the titles from sashelp.vtitle, separate the TFL identifier from the title, and append a record to a text file such as:

```
Figure 16.2.5-1.2a|Individual plasma concentration-time profiles (FAS)
Figure 16.2.5-1.2b|Individual plasma concentration-time profiles (Per protocol)
(etc.)
```

The text file can then be used to load TOC entries; although a bit clumsy, this does allow existing information to be exploited, avoiding the need to hard-code and ensuring that the titles that appear in the TOC match what appear in the figures themselves.

CREATING A BOOKMARKED COLLATED FILE: RTF AND PDF FILE FORMATS

The process for creating a collated file is really just an extension of that for creating a TOC, the main addition being that of iterating through the file list and inserting each file into the collated document, separated by a page/section break. The first page of each file must contain a bookmark, which the TFL identifier in the TOC is then hyperlinked to, so this is also added as a matter of post-processing.

The major steps are as follows:

1. Capture the list/collection of RTF output files
2. Parse TFL IDs out of the file names and load TFL ID/file name combinations to an array, and (optionally) re-order by TFL ID
3. Open each RTF file in Word and insert a bookmark on the first page (then save and close)
4. Open the collated file in Word as a blank document
5. Build the TOC in the collated file as described previously, but without setting hyperlinks (yet)
6. Iterate through the list of RTF output files
7. Below the TOC, insert each RTF file, followed by a page-break and section-break
8. Starting again from the beginning of the RTF file list, do a search/select in the TOC for each TFL ID
9. Set a hyperlink on the selected TFL ID to the corresponding bookmark set in step 3

Many of these tasks have already been illustrated in previous examples, but the following should give some quick insight into the Vbscript syntax required for the new steps:

3. Open each RTF file, insert a bookmark on the first page (on the first TFL identifier, such as 'Table 1'), save and close. Note that the bookmark is named to match the file name without the file extension (i.e. 't_1').

```
wd.documents.open <directory\file reference>
wd.Selection.Find.execute <TFL identifier>
wd.ActiveDocument.Bookmarks.Add <file name>, wd.Selection.Range
wd.activedocument.save
wd.activedocument.close
```

7. Below the TOC, insert each RTF file followed by page/section breaks (remember, break type constants can be looked up on the object browser in Word)

```
wd.Selection.InsertFile <directory\file reference>
wd.selection.insertbreak 3
wd.selection.insertbreak 2
```

8. Search/select each TFL ID in the TOC

```
wd.Selection.Find.execute <TFL identifier>, , , , , , 1, 1
```

9. Set a hyperlink from the TFL ID in the TOC to the corresponding bookmark

```
wd.ActiveDocument.Hyperlinks.Add wd.Selection.Range, ,<file name minus
extension>, , <TFL identifier>
```

The full script is provided with the demo described in Appendix 3.

Collated PDF files

If clients require collated files as PDF documents, it's much simpler to start by assembling a collated RTF file (with bookmarks and hyperlinked TOC) and then write this file to PDF using Acrobat Distiller (which will retain & honor bookmarks and hyperlinks set in the RTF file) than it is to try and assemble a collated PDF file from individual PDF files. The latter is simple enough to do via the user interface to Acrobat Distiller *if a hyperlinked TOC is not required*. If a hyperlinked TOC *is* required, either Acrobat Javascripting or some other PDF-authoring tool will be required. Unlike with MS Word, Acrobat does not offer a macro-building facility. There's no way of translating the steps you would perform by hand through the user interface into Acrobat Javascript other than by first cultivating an understanding of Acrobat Javascript (an admirable undertaking, but not the 'path of least resistance' we're striving for here).

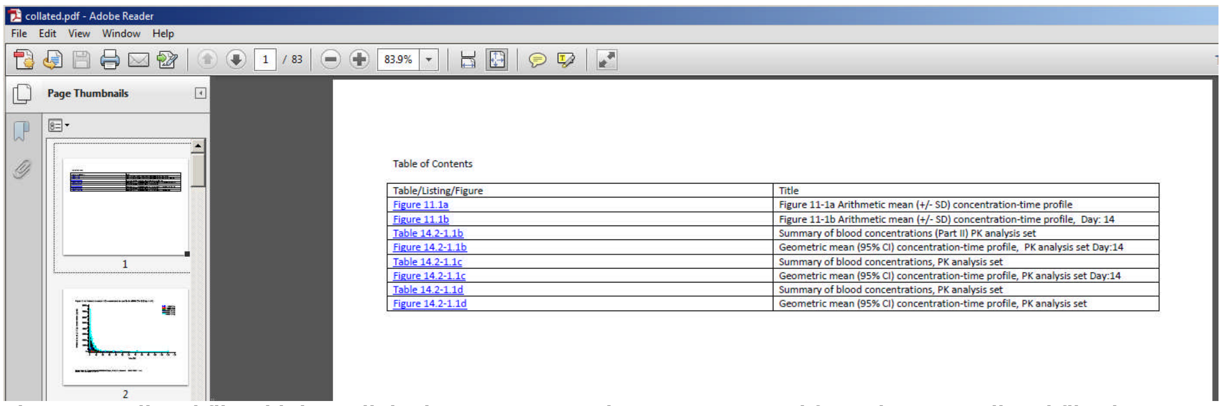


Table of Contents

Table/Listing/Figure	Title
Figure 11.1a	Figure 11-1a Arithmetic mean (+/- SD) concentration-time profile
Figure 11.1b	Figure 11-1b Arithmetic mean (+/- SD) concentration-time profile, Day: 14
Table 14.2-1.1b	Summary of blood concentrations (Part II) PK analysis set
Figure 14.2-1.1b	Geometric mean (95% CI) concentration-time profile, PK analysis set Day:14
Table 14.2-1.1c	Summary of blood concentrations, PK analysis set
Figure 14.2-1.1c	Geometric mean (95% CI) concentration-time profile, PK analysis set Day:14
Table 14.2-1.1d	Summary of blood concentrations, PK analysis set
Figure 14.2-1.1d	Geometric mean (95% CI) concentration-time profile, PK analysis set

Figure 7. Collated file with hyperlinked TOC as a PDF document, created from the RTF collated file via Acrobat Distiller.

CREATING A COLLATED EXCEL WORKBOOK WITH HYPERLINKED TOC

Although less common than RTF or PDF file formats, statistical output sometimes has to be provided in some form of Excel-readable output, but following similar conventions in style of presentation as with more typical RTF/PDF statistical summaries (i.e. using PROC REPORT, as opposed to the unformatted output from PROC EXPORT). Use of ODS HTML to produce Excel-readable output has more recently given way to ODS tagsets.excelxp, which produces markup and style definitions as XML.

Without getting into the finer details of working with tagsets.excelxp, suffice it to say that by wrapping ODS tagsets.excelxp statements around PROC REPORT:

```
ods tagsets.excelxp file="t_1.xls" style=styles.sasweb
    options(embedded_titles='yes' embedded_footnotes='yes'
            Default_Column_Width = "20,12,8,8,8"
            sheet_label="Table 1");

proc report data=final ...;
  --etc.--
run;

ods tagsets.excelxp close;
```

, an Excel-readable tabular summary can be created that is comparable to that produced using the more common ODS RTF destination:

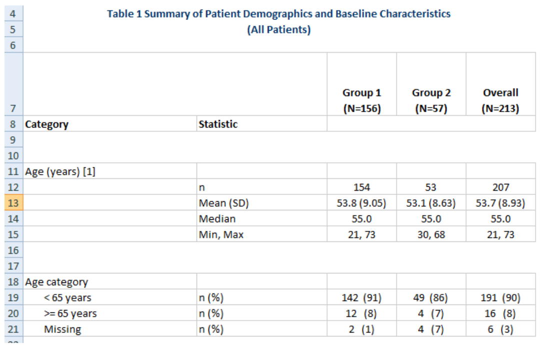


Table 1 Summary of Patient Demographics and Baseline Characteristics
(All Patients)

Category	Statistic	Group 1 (N=156)	Group 2 (N=57)	Overall (N=213)
Age (years) [1]				
	n	154	53	207
	Mean (SD)	53.8 (9.05)	53.1 (8.63)	53.7 (8.93)
	Median	55.0	55.0	55.0
	Min, Max	21, 73	30, 68	21, 73
Age category				
< 65 years	n (%)	142 (91)	49 (86)	191 (90)
>= 65 years	n (%)	12 (8)	4 (7)	16 (8)
Missing	n (%)	2 (1)	4 (7)	6 (3)

Figure 8. Summary table in Excel-readable format, created via PROC REPORT and tagsets.excelxp

In the same way that a hyperlinked TOC and/or collated file makes it easier to navigate large batches of output in RTF or PDF format, the Workbook/Worksheet organization of Excel files makes them convenient for reviewing batches of summary tables. Although the tagsets.excelxp destination allows for the generation of multi-sheet workbooks within a single set of ODS statements, there is no inherent or convenient way of appending individual, separate tables (i.e. those created by several different table programs) into a single workbook. However, since the XML output that is generated by tagsets.excelxp can be very easily manipulated as simple text, individual files can be concatenated into a single workbook via some fairly manageable post-processing.

Essentially, the XML file for each table can be roughly divided into what we might call a 'header', containing style definitions, followed by a worksheet containing the data to be displayed, and the two can be very easily delineated by their respective tags ("<Styles>"/"</Styles>" tags giving the boundaries of the style definitions, and "<Worksheet>"/"</Worksheet>" tags giving the boundaries of the worksheet). So for the table shown above, these sections would look as follows:

Header:

```
<?xml version="1.0" encoding="windows-1252"?>

<?mso-application progid="Excel.Sheet"?>
<Workbook xmlns="urn:schemas-microsoft-com:office:spreadsheet" ...(etc.) >
 ...(etc.)
 <Styles>
  <Style ss:ID="_body">
   <Interior ss:Pattern="Solid" />
   <Protection ss:Protected="1" />
  </Style>
  ...(etc.)
 </Styles>
```

Worksheet:

```
 <Worksheet ss:Name="Table 1">
 ...(etc.)
  <Table ss:StyleID="_body">
```

```
        ...(etc.)
        <Row ss:Height="15" ss:StyleID="_body">
         <Cell ss:StyleID="systemtitle4__c" ss:MergeAcross="4">
          <Data ss:Type="String">
           Table 1 Summary of Patient Demographics and Baseline Characteristics
          </Data>
         </Cell>
        </Row>
        ...(etc.)
        </Table>
        ...(etc.)
       </Worksheet>
       ...(etc.)
      </Workbook>
```

To build the most basic collated workbook, all that is required is to concatenate a header, containing all the necessary style definitions, followed by the worksheet stripped out of each individual XML file, into a single XML file. In other words, we start with everything down to and including the "</Styles>" closing tag, and append to that everything bounded by the "<Worksheet>/</Worksheet>" tags for each table, and then end with the closing "</Workbook>" tag.

Note in the code fragment above that the 'sheet_label' option was specified, which gives a label for each tab in the collated workbook:

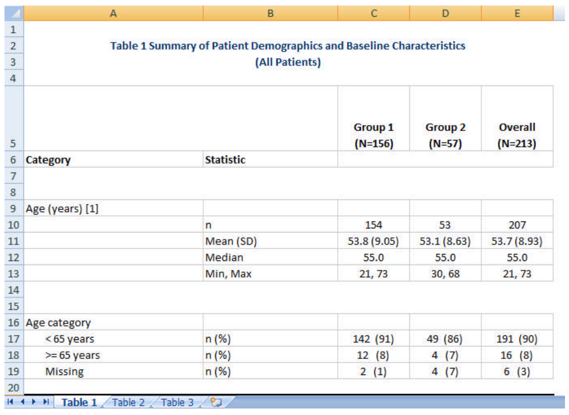


Table 1 Summary of Patient Demographics and Baseline Characteristics
(All Patients)

		Group 1 (N=156)	Group 2 (N=57)	Overall (N=213)
Category	**Statistic**			
Age (years) [1]				
	n	154	53	207
	Mean (SD)	53.8 (9.05)	53.1 (8.63)	53.7 (8.93)
	Median	55.0	55.0	55.0
	Min, Max	21, 73	30, 68	21, 73
Age category				
< 65 years	n (%)	142 (91)	49 (86)	191 (90)
>= 65 years	n (%)	12 (8)	4 (7)	16 (8)
Missing	n (%)	2 (1)	4 (7)	6 (3)

Figure 9. Multi-sheet Excel workbook created by collating XML files.

As with the collated RTF (or PDF) file, a hyperlinked Table of Contents can also be an asset, and this can be inserted as the first worksheet in the workbook:

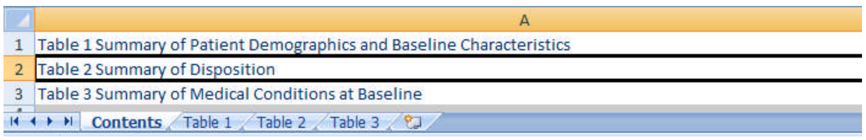


Table 1 Summary of Patient Demographics and Baseline Characteristics
Table 2 Summary of Disposition
Table 3 Summary of Medical Conditions at Baseline

Figure 10. A simple Contents page inserted into the collated workbook.

And like the other worksheet pages, the Contents page is inserted between "<Worksheet>/</Worksheet>" tags:

```
      <Worksheet ss:Name="Contents">
       <Table ss:StyleID="table">
       <ss:Column ss:AutoFitWidth="1" ss:Width="600"/>
       <Row ss:Height="15">
        <Cell ss:StyleID="contentitem" ss:HRef="#'Table 1'!A1">
         <Data ss:Type="String">
          Table 1 Summary of Patient Demographics and Baseline Characteristics</Data>
      (etc.)
```

Note that a link to the corresponding table is provided via the 'HRef' reference, so that clicking on the entry in the Contents page will navigate to the desired workbook tab.

The Contents page can be assembled dynamically during construction of the collated file, since as with the rest of the collated file it really just involves simple text processing. Each individual table (XML file) is read and parsed in order to strip out the worksheet, so it takes very little extra work to identify and pull out the title lines as well, which are then placed between the appropriate tags and assembled into the table that comprises the TOC.

An example macro 'mworkbook.sas', along with some sample output files created using tagsets.excelxp, is provided with the demo described in Appendix 4. 'mworkbook.sas' creates a collated workbook from individual output files, including a simple hyperlinked TOC as the first worksheet.

CONCLUSION

If you've tended to work on projects with deliveries that consist of small numbers of output files, you may have been lulled into thinking that the same techniques for managing output can be applied when you hit a project with large numbers of output files. You'd probably be in for a shock though the first time you had to put together a delivery of 400 TFLs, when confronted with the task of confirming that all required files are present and that no 'stray' files are mixed in, and that all files are 'current' and validated. You'd probably be further dismayed if you discovered in the week prior to the delivery that the client was expecting all output to be collated into a single file, complete with a hyperlinked Table of Contents, especially once you'd started in on assembling this file manually.

Hopefully the issues discussed in this paper have been useful in figuring out how to plan for a delivery consisting of large numbers of output files, and the technical solutions for several of the automatable tasks will prove useful, at least as starting points for adaptation, and this will help mitigate the otherwise daunting task of managing large batches of output.

ACKNOWLEDGMENTS

The authors would like to thank our employer inVentiv Health Clinical for supporting our participation in PharmaSUG, along with all our friends at inVentiv, especially Angela Ringelberg for encouraging creativity and sharing of ideas.

REFERENCES

Brian Fairfield-Carter and Stephen Hunt, *Fast and Efficient Updates to Project Deliverables: The Unix/GNU Make Facility as a Cross-Platform Batch Controller for SAS®*, PharmaSUG 2008, SGF 2008

Stephen Hunt, Tracy Sherman and Brian Fairfield-Carter, *An Introduction to SAS® Applications of the Windows Scripting Host*, PharmaSUG 2004, SUGI 30 (2005)

CONTACT INFORMATION

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

Name: Brian Fairfield-Carter
Enterprise: inVentiv Health Clinical
E-mail: fairfieldcarterbrian@gmail.com

APPENDICES

The following appendices describe demos that can be downloaded from http://sourceforge.net/projects/shellout/files/ (file "**AD07_PharmaSUG2014_demos.zip**"). Each demo consists of a 'readme' file that briefly explains the point behind the demo and how to run it, along with the sample files and program files. The demos are 'idealized' in the sense that the files the scripts operate on follow fairly simple file-naming and design conventions (and are few in number, but the scripts should function equally well on larger numbers of files), but the hope is that they will serve as a useful starting point for adapting to specific project requirements.

In demos 1, 2 and 3, the script files are called '_run_me.vbs', and are executed simply by double-clicking. The scripts can be modified by editing in a text editor (such as Notepad), and each script includes comments which will hopefully help in identifying specific functionality that you might want to adjust. Demo 4 requires SAS, since the central element is 'mworkbook.sas', which builds a collated Excel-readable file by concatenating individual files created via tagsets.excelxp.

APPENDIX 1: CHECKING FOR A 1:1 MATCH BETWEEN PROGRAMMING PLAN AND OUTPUT

Folder: demo1

Files:

1. _readme.txt: brief notes explaining the demo
2. programming_plan.xlsx: example programming plan listing table/figure/listing identifiers, titles, output file names, etc.
3. TFLs.txt: comma-delimited text file created by saving the first three columns from 'programming_plan.xlsx' in comma-delimited format from Excel
4. \TFLs\t_1.rtf, t_2.rtf, t_3.rtf, t_15.rtf: example tables that might make up a batch of statistical output
5. _run_me.vbs: script file to be executed by double-clicking

Output: the '_run_me.vbs' script compares the files listed in 'TFLs.txt' against the files contained in the 'TFLs' sub-directory, and creates a discrepancy report 'discrepancies.txt' showing the file name and title of any file listed in the programming plan that does not exist in the 'TFLs' sub-directory, and the file name of any file in the 'TFLs' sub-directory that does not appear in the programming plan.

Adaptation: if your statistical output is in a different file format (i.e. .lst, .pdf, .doc, etc.), alter the file-extension filter:

```
if UCase(right(filename,4))=".RTF" then
```

You may need to allow for commas in the titles, and if so, consider using a different delimiter (such as a pipe) in the text file, which will require updates to these lines:

```
file_array_frmfile(line_count,0)=split(text_,"|")(2) & ".rtf"
file_array_frmfile(line_count,1)=split(text_,"|")(0)
file_array_frmfile(line_count,2)=split(text_,"|")(1)
```

APPENDIX 2: COMPARISON OF PRIOR AND CURRENT PRODUCTION OUTPUT

Folder: demo2

Files:

1. _readme.txt: brief notes explaining the demo
2. \PRIOR\t_1.rtf, t_2.rtf, t_3.rtf: example statistical output, previous (archived) version
3. \CURRENT\t_1.rtf, t_2.rtf, t_3.rtf, t_15.rtf: example statistical output, current version
4. _run_me.vbs: script file to be executed by double-clicking

Output: the '_run_me.vbs' script opens each RTF file in the 'CURRENT' sub-directory and does a Word-comparison against the file (if any) in the 'PRIOR' sub-directory sharing the same file name, and writes a summary of differences to 'compare_log.txt' (as well as noting any files in the 'CURRENT' folder which do not have a corresponding file in the 'PRIOR' folder).

Adaptation: assumes that files are RTF, but if they're in some other Word-readable format (i.e. text or .doc), you'll need to adjust the file-extension filter:

```
if fso.GetExtensionName(file) = "rtf" then
```

If run date/time are displayed somewhere other than in a footnote containing the text "Source code:", if you want to avoid reporting those discrepancies you'll need to modify these lines with a rule that will identify the offending lines:

```
myRange.SetRange myRange.Start-121, myRange.End
if instr(myRange,"Source code: ") then
```

```
      ' DIFFERENCES IN THE SOURCE CODE FOOTNOTE ARE IGNORED!
      ' MsgBox(myRange)
    else
```

Hint: un-commenting the 'MsgBox(myRange)' line will give you a dialog displaying the selected section of text, which will be helpful in adjusting the code to identify and trap the text containing run date/time.

APPENDIX 3: CREATING A HYPERLINKED TOC AND/OR COLLATED FILE

Folder: demo3

Files:

1. _readme.txt: brief notes explaining the demo
2. f_1.rtf, f_3.rtf, t_1.rtf, t_1_1.rtf, etc.: example statistical output, following simple file-naming conventions
3. _run_me(unordered).vbs: script file to be executed by double-clicking; doesn't order RTF files
4. _run_me(ordered).vbs: alternate script file, which does order RTF files according to TFL number

Output: the '_run_me(unordered).vbs' script creates a file 'TOC.doc' which will be either just a Table of Contents with each entry hyperlinked to the corresponding RTF file (if the 'file_hyperlink()' function is activated - see below under 'Adaptation'), or will be a collated file prefaced with a Table of Contents where each entry is hyperlinked to the corresponding bookmark in the collated file (if the 'collate_and_hyperlink()' function is activated), and in either case entries will follow the default (alphabetical) ordering by RTF file name. The '_run_me(ordered).vbs' script will produce the same, except that TFL number is parsed out of the file name, and entries will follow numeric ordering by TFL number. If TOC.doc is converted to PDF via Acrobat Distiller, the hyperlinks and bookmarks will be honored, and will function in the PDF file.

Note that if the 'collate_and_hyperlink()' function is activated, copies of each RTF file are created in a 'temp' sub-directory, since each RTF file has to have a bookmark inserted on the first page. For instance, if you open one of the files in the 'temp' sub-directory and go to the 'Insert Hyperlink' dialog, you should see something like this:

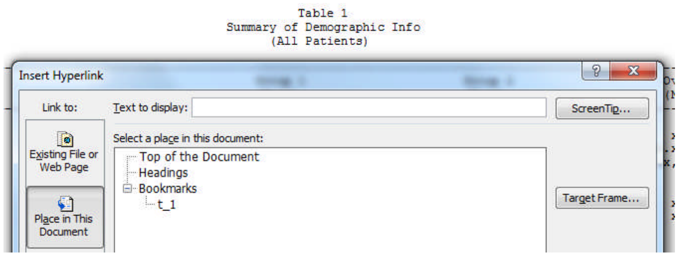


Figure 11. Temporary copy of an RTF file, showing the added bookmark.

In other words, a bookmark (in this case 't_1', derived from the file name) has been added to the file, and the entry for this table in the TOC will by hyperlinked to this bookmark when the file is added to the collated file.

Adaptation: switching between a TOC and a collated file is fairly low-tech, and just involves manually commenting/un-commenting the appropriate function calls (i.e. by editing the script in Notepad):

```
'file_hyperlink()
collate_and_hyperlink()
```

In this case, 'file_hyperlink()' is commented out in favor of 'collate_and_hyperlink()', meaning that a collated file prefaced by a TOC will be created as 'TOC.doc'.

Methods for capturing TFL number and title(s) out of the individual RTF files are obviously pretty sensitive to the particular style conventions used in the output files.

```
wd.Selection.MoveDown
wd.Selection.MoveDown
wd.Selection.MoveDown
wd.Selection.MoveDown ,,1
Identifier=wd.Selection
```

This literally takes the cursor from the top of the document down a set number of lines to whatever is the 'standard' location for title lines containing TFL number and descriptive titles (i.e. if you open t_1.rtf, you'll see that this moves past the protocol-identifier and blank title line, down to the 'Table 1' line, then moves down another line while extending a selection, thereby grabbing the 'Table 1' title line). This will clearly require modification if your output files follow other stylistic conventions.

Similarly, capturing the TFL number used to order files in the TOC.doc output (when using the '_run_me(ordered).vbs' script) is sensitive to file-naming conventions. Without going into great detail, note that in the section of the script prefaced by the comment

```
        ' NEXT, LOAD ORDERING NUMBER DERIVED FROM .RTF FILE NAME
```

, there are a number of 'MsgBox' calls commented out -- these have been strategically placed so that by un-commenting them, you'll be able to check on what the ordering number looks like for each file under your particular file-naming conventions, and make (and check) any necessary adjustments. To provide further illustration, a sub-directory "alternative_filenames" has also been provided, where file naming conventions have been changed slightly, and the 'ordering number' section of the script modified slightly to accommodate.

As a final hint, if you adapt this script to your specific requirements, do your initial modifications and testing on a small number of files, and only increase the number of files once you have things working to your satisfaction. It also works better to do Vbscript development on your local desktop (i.e. rather than on a Citrix desktop), primarily because if anything goes wrong with your script alterations, you have far more control over shutting down stray processes (i.e. you can go to your local 'task manager' and kill stranded Word instances, as well as stranded instances of the Windows Scripting Host ('wscript.exe'), things you may not be able to do on a Citrix desktop).

APPENDIX 4: CREATING A COLLATED EXCEL WORKBOOK, WITH HYPERLINKED TOC

Folder: demo4

Files:

1. _readme.txt: brief notes explaining the demo
2. t_1.xls, t_2.xls, t_3.xls: sample statistical output, created via PROC REPORT and ODS tagsets.Excelxp
3. header.xml: the 'header' section (style definitions) stripped out of one of the 't_1.xls' file
4. mworkbook.sas: the SAS macro that builds the collated workbook
5. xml_to_xls.vbs: an optional script to open the collated workbook (as XML, created by concatenating the 'header' and the various worksheet pages) in Excel, and re-save it as a 'native' Excel file.

Output: since this demo requires SAS, the output to 'mworkbook.sas', the collated Excel workbook 'collated_workbook.xls', is provided. This file is created by concatenating 'header.xml' to a 'contents' worksheet (where entries on the contents page consist of titles pulled out of the 't_1.xls', 't_2.xls', etc. files, with hyperlinks set as style attributes), followed by the worksheets stripped out of the 't_1.xls', 't_2.xls', etc. files.

Adaptation: retrieving and ordering the source files is not particularly fancy (but could be modified to use a directory listing akin to that used in the log-check macro); set 'DIR' to the directory containing the tables to be collated (i.e. "F:\projects\ABC\output"), and use 'FILELST' to list tables in the order they should appear in the TOC and in the collated file:

```
        %LET DIR=%str(<DIRECTORY>); *** (ENTER YOUR WORKING DIRECTORY HERE);
        %LET FILELST=t_1 t_2 t_3;
```

Depending on the specific conventions used for titles, the block of code that captures the title lines out of each table may need to be adjusted:

```
        if index(str_,'ss:StyleID="systemtitle') & index(upcase(str_),">TABLE ") then
        do;
```

Finally, note that the 'header' contains style definitions referenced by all the concatenated files, so be aware that depending on which file you take the header from, you may need to supplement it with style definitions pulled out of other tables, to make sure that tables in the collated file end up looking exactly as they do in the individual output files.