ABSTRACT
Regardless of whether an organization creates simple or complex SAS programs, a controlled programming environment should embrace the concepts of the software development life cycle (SDLC). To do this, processes need to be spelled out and tools put in place that support the phases of activity specified in SDLC models. SAS programming for clinical trials has some unique characteristics when compared to software development projects in general. This makes the use of full-featured enterprise-level configuration management systems impractical. This paper demonstrates an approach to constructing a rudimentary technical infrastructure to support the clinical trials programming life cycle. Topics addressed include the promotion of code from development through to production, versioning, change control, source control, and security. The techniques presented are applicable to a Windows-based SAS environment.

INTRODUCTION
Large software systems are developed by multi-person teams employing a software development lifecycle (SDLC). Following a SDLC is meant to provide structure for a project, facilitate coordination and communication among team members, and minimize mistakes leading to do-overs.

In clinical trials programming, SAS programs tend to be smaller and a SAS “project” may be worked upon by one person. It is true that macro systems can contain thousands of lines of code and be worked on by teams of programmers, but for the most part, these systems still are smaller and less complicated than a piece of software such as Windows or Excel. Nevertheless in both environments, roughly the same activities are performed to produce a production program. For smaller SAS programs, it happens that the process is often not formalized and may not be readily apparent to the programmer.

Formalizing the programming task requires integrating the SDLC. The benefits are that programs are developed in a consistent manner, there is traceability between program, data, and output, and there is clear documentation of the path taken to realize a task. To work in a SDLC model, an infrastructure needs to be in place to support the various stages of activity. For example, as code is developed and run, many changes occur. As stable versions are produced, it is desired to permanently archive these versions. When code becomes production, new requirements will necessitate change and the cycle starts over. A technical infrastructure to help control this process can be bought out of the box and a process can be wrapped around it. But this can be expensive or too elaborate for a small programming shop. Thus, this paper describes an approach to setting up a rudimentary technical infrastructure. SAS and Visual Basic are used to create some technical tools to supplement operating system features. A Windows-based SAS environment is the foundation.

A BLURB ABOUT THE SOFTWARE DEVELOPMENT LIFE CYCLE
If one studies the principles of software programming, ideas and theories can go quite deep. In general, the software development life cycle is a framework specifying a series of activities conducted in an orderly fashion that takes a concept through to retirement of a software system. Its goal is to successfully deliver a piece of software, whether it be a large system or something smaller. The sequence of activities can vary depending on the particular SDLC model employed. The various “models” are beyond the scope of this paper. However, the phases of a SDLC are fairly well defined as:

1. Requirements identification
2. Planning and design to meet the requirements
3. Programming (Coding)
4. Testing
5. Production use
6. Maintenance

Documentation is important at each stage. And each phase has its own best practices on achieving deliverables.
Naturally, a technical infrastructure is needed to support this framework and more importantly, from a programmer's point of view, enable controlled development of the code. The infrastructure ensures that teams can work effectively and efficiently. By having a structured organization, software projects can be built by one group of people, and be maintained by others. This is important in today's business world where personnel changes occur often.

LARGE SOFTWARE DEVELOPMENT PROJECTS
Large software projects involve multi-person and geographically dispersed teams whose end-result is a single product. For example, imagine the programming team at Microsoft that produces the Excel spreadsheet. I contrast this with a single-run SAS program written by a statistician that manipulates some data and runs a PROC GLM. This is also considered a "software product", but on a smaller scale. Of course, there are more complicated macro systems; and one could view a complete clinical trial analysis as one software project. However, these may still pale in comparison to the sheer magnitude of code and number of developers for a large software development project.

In a larger project, the requirements identification can be a long involved process, consisting perhaps of evaluation of current systems and processes, interviews with users and customers, interaction with vendors and seeking management buy-in. Documentation is the primary result of this phase. During the subsequent design phase, high-level schematics and technical specs are written, resulting in more documentation. A system is divided into modules to facilitate assignment of programming tasks.

At the coding stage in multi-person teams, different programmers will be assigned different modules. When all is ready, the whole set of modules is compiled into one and put into testing. Documentation of testing is required. When bugs are ironed out, the software is put into production. The maintenance phase begins, with all changes carefully controlled and going through the SDLC again.

To manage this process, commercial "configuration management" systems are employed to enable coordination and control among geographically dispersed, multi-person teams. The system acts as a police or regulator, including features to prevent overwriting of documents and code. Different programmers can each upload their assigned pieces of code and the SDLC system will gather it all and compile it, sending error and problem messages to the appropriate programmer. Commercial products are full-featured, require manpower to run them, and costs perhaps tens of thousands of dollars. Repositories for these systems require an industrial strength database. Implementing a system like this is a project in its own right. Example products include Rational's ClearCase, Perforce, and Serena PVCS. I do not have first-hand experience with any of these products so this paper will not delve into the details of these systems.

CHARACTERISTICS OF CLINICAL TRIALS PROGRAMMING
SAS is the de-facto programming language used for clinical trials analysis. Other statistical programming languages might be used, e.g. S-PLUS. SAS programs vary in size and complexity. They range from short programs written for ad-hoc analyses to big macro systems for producing a whole set of output. In addition to analyzing clinical trials data, SAS programs can also be created for a variety of data processing and reporting needs. An example of a SAS "project" is an analysis of data from a clinical study.

Biostatistics staff will have expertise in statistical programming and not code management systems. Departments usually cannot afford to hire someone to administer a source control system. System maintenance might be delegated to the IT department. Except for large macro systems or clinical studies, it is likely that one programmer works on a project. A SAS program tends to be self-contained, i.e., not many components or modules, unless it's an entire macro system.

The source, i.e. files to be controlled, also includes output from running SAS programs. This includes logs, listings, tables, and graphs. I compare this with a software project where only the code and associated documentation are the items to be controlled.

Processes are in place governing programming standards, code and output review, archival, and required documentation. Requirements may be as simple as a sketch mock-up by a statistician or be a more formal document, e.g. analysis plan. Design can be as simple as commenting code or be a more formal specifications document. Unless the organization is larger and mature, change often happens "on the fly", sometimes with no trail left showing what was previously done. This behavior is often driven by the urgency of the request.
For big macro systems, the process may resemble what is done for a large software project. There are discussions with different people to define requirements for generalized software. Documentation may be extensive, multi-person teams may be employed, and programs thoroughly validated prior to use. For the most part, the SDLC is followed for these projects.

Because SAS source includes both code and output, a primary weakness of commercial source control systems is the lack of capability to run a program within them. Programs must be checked-out, run, and then checked-in, along with output. This lack of integration with the natural SAS directory-based environment may lead to uncontrolled programming. Of course, there are some more specialized products that have appeared on the market (e.g. Waban) but these can require elaborate administration.

INTEGRATING SDLC INTO CLINICAL TRIALS PROGRAMMING

To meet 21 CFR Part 11 requirements for program validation, it’s necessary to have a controlled programming environment. Protocols, analysis plans, programming specifications and mock tables represent output from the planning stage and need to be safely stored. Before these are finalized, they go through many versions and involve multiple people. Summary tables, listings, and graphs need to be linked to the programs that generate them. The datetime stamps appearing on files and printed within output should demonstrate clearly the timeline used to prepare the study report. In this manner, it’s possible to trace a specific analysis to the program and data used to produce it. And if multiple versions of analysis output are floating around, as it sometimes happens, then it is possible to identify the data and program used to generate it. Sometimes it is necessary to revert to earlier versions. The entire package of related materials needs to created in an orchestrated manner and documented so that an external auditor will readily be able to verify that a drug sponsor’s results are trustworthy. These considerations all point toward the adoption of SDLC processes and tools to ensure robust electronic records.

COMPONENTS OF A SDLC INFRASTRUCTURE

Standardization is a key concept. To adopt a SDLC for SAS program development, it is necessary to have standard folder structures and rules dictating where to store what. To allow for program validation and version control, separation of environments for development, testing, and production is needed. There must be a process to move files upward from development to testing and ultimately production. This “file promotion” should be documented. Information to be captured includes the name of the file promoted, where it is promoted, who promoted it, and the date of promotion. When a file is made production, it must be versioned so that in the future a particular output can be traced to a particular set of source code as well as serve as a starting point for change. The file in production should be protected to prevent deletion. Folders should be provisioned for storage of code, results, and documentation.

Let’s discuss further about a standard folder structure because this is the foundation. For example, a multiple level folder hierarchy might begin with the project directory, containing a folder for each clinical study. The third level might name the type of analysis, e.g. Interim, Final, or Ad-hoc. From this level on, folder names would be limited to a pre-approved list. For example, having three areas, i.e. DEVEL, QA, PROD, would model the flow of programs through phases of development.

Pictorially, this looks like:

<table>
<thead>
<tr>
<th>Folder Name</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>SAS</td>
<td>Programs, logs, listings</td>
</tr>
<tr>
<td>TAB</td>
<td>Output tables</td>
</tr>
<tr>
<td>SSD</td>
<td>Raw data</td>
</tr>
<tr>
<td>SSD_D</td>
<td>Derived data</td>
</tr>
<tr>
<td>DOCS</td>
<td>Documentation such as analysis plans</td>
</tr>
</tbody>
</table>
Putting it all together, our picture looks like:

![Folder Structure Diagram]

To make it easier to set up standard folders each time a clinical study starts, there needs to be a tool that guides the creation of folders and applying of a correct set of permissions. Promoting files between folder hierarchies can be architected as a simple copy-and-paste. However, to ensure file security and versioning, the DEVEL-QA-PROD hierarchy should be protected more strictly at higher phases. Furthermore, to enable versioning and auditing of user actions, we see that tools are needed. In the next section, we describe a possible technical implementation in more details.

**AN IMPLEMENTATION USING SAS AND VISUAL BASIC IN A WINDOWS SERVER ENVIRONMENT**

The goal is to build a rudimentary technical infrastructure to facilitate adoption of a SDLC for SAS programming. We will use basic elements of SAS, Visual Basic scripting, and Windows itself. We are assuming a Terminal Server environment where users log into the server to do their work.

To facilitate users’ consistent setup of standard folder structures when beginning an analysis, a tool is built to allow users to create appropriate folders. The tool communicates with a listener that does the actual work of creating folders and applying appropriate access permissions.

As mentioned, we need access controls on certain folders so that files cannot be accidentally deleted. Our philosophy is to prevent accidents rather than a strict lockdown. Imposing read-only permissions on PROD folders conforms to the philosophy. Once a file is considered production and versioned, it cannot be deleted. However, this means that there must be a means for a user to promote a file into the PROD directories.

File promotion is the act of copying a file to the next higher phase, whether QA or PROD. A tool is built to communicate with a listener that does the actual work of copying files and versioning them. The exact nature of the listener is described in the next section.

During the phases of the SDLC that generate documents, output such as requirements and design specifications (e.g. analysis plans, functional outlines, etc.) are worked upon in the DEVEL folders and then promoted into PROD. As the coding and testing phase begins, programmers refer to the PROD version of documents. Statisticians work on the DEVEL version and promote them when ready.

QA folders are the intermediate level where code is promoted when ready for peer review or testing. Peer programmers can work in the QA area to extensively test the programs. When everything is done, programs are promoted to the PROD level and run as production.

Because a user cannot write logs into protected folders, a tool is built to enable execution of SAS programs and utilities in these directories.
During the maintenance phase of the SDLC (e.g. revision of analysis plan specifications), production versions of programs can be manually copied back to the DEVEL level and the development process is repeated. This process covers change control requirements.

To summarize, the technical implementation includes a standard folder structure and five utilities. (1) A script that guides creation of standardized folders. (2) A listener that creates the folders and applies access permissions based on pre-determined rules (e.g. if folder name is “randomization”). (3) A script that allows promotion of files between different folder levels, i.e. promote tool. (4) A script that allows running of programs and utilities in read-only directories. (5) A listener that copies files or executes them, creates versions, and writes to audit trails.

**EMPLOYING SOCKETS TO BUILD A LISTENER**

In order to implement a folder structure consisting of read-only folders and still enable users to promote files from unprotected areas to protected areas and also run programs, we need to devise a method to “jump the hurdle” between these areas. This is where the socket comes into play.

The FILENAME statement supports a SOCKET access method. A socket is a communications link between two applications. Thus, one SAS program can act as a client by using the socket access method. And the other can act as a server. Since we have two processes going on, each process can run under a different security context.

Our approach to both the file promotion and folder creation listeners utilizes a third-party account that is set up to have read/write access to protected directories. Users do not know this account’s password. When called, the new folder listener creates the requested folders with this account being the owner and permissions allowing read/write in protected areas. In unprotected DEVEL directories, all users have read/write access.

There are two listeners at work, one to support the folder generation, and the other to handle file promotions and running programs. The “New Folder” listener has the following logic flow:

(1) Loops and waits for a client request by listening on port 6000.

```sas
filename protect socket ':6000' server;
%macro loop;
  %do %while (1);
  data _null_;
    length dirpath $ 150 dirpgm $ 100;
    infile protect lrecl=200 pad end=eof;
    …
  %end;
%mend loop;
%loop;
```

(2) Executes the program named in the DIRPGM variable under the third-party account’s security context to create the folders. Refer to the newfold_12345.sas code below to examine the structure of this program.

```sas
call execute ('%' || "include '" || trim(dirpgm) || "'');
```

(3) Examines the root folder named in the DIRPATH variable and if a special folder (e.g. randomization data), applies added restrictions. If not a special folder, a general set of permissions is applied, i.e. read-only on PROD directories for all users. For this task, regular expressions are used to test each directory path named in the DIRNAME variable. We use the Windows CACLS command to set access permissions.

```sas
dir1 = "E:\\S+\\(p|P)(r|R)(o|O)(d|D)"
... regexp = "/(" || dir1 || ")*/";
rel = prxparse(regexp);
... proddir = prxmatch(rel, dirname);
... if proddir = 1 then do;
  CALL system ('cacls ' || trim(dirname) || ' /e /p COMPUTER\Users:R');
End;
```
The file promotion listener has a similar logical structure to handle functionality such as copying files, creating versions, writing audit trails, and running programs. If files are not successfully promoted, e-mail and messages get generated. Port 6001 is used. Some code excerpts follow:

(1) Copies files as part of requested file promotion. SOURCEFILE contains the path and name of the source file. DESTFILE contains the path and name of the destination file.
   
   if action = "C" then do;
   call system ("COPY " || trim(sourcefile) || COPY || trim(destfile) || COPY);
   
(2) If promoting into a PROD directory, versions file and writes to audit trail. Below a new audit dataset is created in the library ARCHLIB. Then the first version is archived with a "_1" appended to the file name (represented by variable ARCHIVEFILE).
   
   IF rc = 0 THEN DO;
   rc = libname ("archlib", trim(archpath));
   rc = exist ("archlib.audit");
   END;
   if rc = 0 then do; *** does not exist ***;
   *** construct audit dataset and add first entry – version file ***;
   put "libname ssd_d " archpath +(-1) "\";
   put "proc sql noprint ;"
   put "create table ssd_d.audit";
   ...
   put "data _null_;";
   put 'call system ("COPY " destfile +(-1) "\" archivefile +(-1) "\"");';
   put "run;";
   put "proc sql noprint ;";
   put "insert into ssd_d.audit";
   <%include is used to execute the code that is generated>

(3) Uses the EMAIL access method to send messages to users if file promotion fails as indicated by the flag variable FAILED that has been computed for each file.
   
   FILENAME extrmail EMAIL TO="&promoter_email" SUBJECT="&promotion_status";
   
   ** Suppress page breaks. **;
   OPTIONS formdlim = ' ';
   PROC printto PRINT = extrmail;
   RUN;
   
   DATA _NULL_;
   FILE PRINT NOTITLES;
   SET list_of_files;
   LENGTH status $20;
   IF failed = 1 THEN status = 'Not Copied:';
   ELSE status = 'Copied:';
   PUT @1 status @13 pgm;
   RUN;
   
(4) Listens for requests to run programs in protected directories. The following code handles files with a ".SAS" extension and sends the request to a batch scheduler.
   
   if upcase(fileext) = "SAS" then do;
   put "data _null_;";
   put 'file "<batch queue directory>_sas_run_job_<unique id>\.bat";';
   put "put 'cd /D " currentpath +(-1) "\";";
   put "put 'sas -rsasuser -noicon -nosplash -sysin "\" filename_ \";";
   put "put 'msg " user "/time:5 Job " filename_ " has completed.';";
   end;
   <%include is used to execute the code that is generated>
The SOCKET server will wait for requests each time the %DO loop returns to the top. Thus the listener continues functioning unless the process is killed. Only one request can be handled at a time, but since file promotions and folder creation happen at occasional intervals, there is not a problem with excessive queuing. In the next section, we examine the nature of the clients that “speak to” the listeners.

CLIENTS WRITTEN WITH VISUAL BASIC SCRIPTING
Visual Basic Scripting Edition ("VBScript") is a scripting language built into Windows. Scripts written in VBScript get executed by the Windows Script Host that is installed by default on a Windows server. The language is a subset of the full Visual Basic language used for applications development. It has a slightly different syntax. Scripts can be developed using any text editor. As with any scripts, they can be simple or complex. Scripting using VBScript is akin to shell scripting in Unix environments.

In our context, we use VBScript to create command-line-based clients (“front ends”) for users to interact with the listeners. These utilities run under the security context of the user invoking them. Thus the scripts themselves cannot write anything within protected directories. Instead, the scripts are code-generators, writing short SAS programs that use the FILENAME SOCKET access method in client mode to communicate with the SAS listeners running under a third-party account.

We will illustrate how the “New Folder” VBScript front-end performs a few functions. The utility is accessed by right clicking on a folder name and then selecting the tool from the Send-To menu.

(1) Initialize shell object and retrieve current directory name
   Set WshShell = WScript.CreateObject("WScript.Shell")
   currdir = WshShell.CurrentDirectory

(2) Checks that a directory was chosen, and not a file. DNAME contains the path that was clicked by the user.
   If Not fs.FolderExists(dname) Then
      MsgBox Trim(dname) + " is not a directory!", 16
      WScript.Quit
   End If

(3) Asks user if special folders needed
   WScript.stdOut.Write "Create \PROD\PDF? Yes \[Y\], No \[N\]? : 
   create_opt = Wscript.stdIn.ReadLine
   create_opt = UCase (create_opt)
   If create_opt = "Y" Then
      prodpdf = True
   End If

(4) Generates a SAS program (e.g. temp_12345.sas) that contacts the listener, passes information about the program to run to create folders, and asks the listener to apply the correct permissions
   filename cp socket 'computer:6000'
   data _null_;
      file cp;
      put 'FILEPATH=E:\PROJECT\STUDY';
      put 'PGMNAME=E:\TEMP\newfold_12345.sas';
   run;

   The newfold_12345.sas program has the following form based on standard folders and user-selected special folders:
      data _null_;
      infile cards ls=200 truncover;
      length foldname newdir $ 200;
      input fullpath $200.
      *** create the directory if it does not exist;
      foldname = reverse(scan(reverse(fullpath),1,'\'));
      fullpath = substr(fullpath,1,length(fullpath)-length(foldname));
      newdir = dcreate(foldname,fullpath);
cards;
E:\PROJECT\STUDY\FINAL_REPORT\DEVEL
E:\PROJECT\STUDY\FINAL_REPORT\DEVEL\DOCS
E:\PROJECT\STUDY\FINAL_REPORT\DEVEL\SAS
E:\PROJECT\STUDY\FINAL_REPORT\DEVEL\SSD
E:\PROJECT\STUDY\FINAL_REPORT\DEVEL\SSD_D
E:\PROJECT\STUDY\FINAL_REPORT\DEVEL\TAB
...
E:\PROJECT\STUDY\FINAL_REPORT\PROD\PDF
;
run;

The code is executed via a brief SAS session and no log is kept.
  WshShell.Run "sas -nolog -nosplash -rsasuser -sysin temp_12345.sas", 7, True

The file promotion and file run clients are similarly scripted. Highlights of the file promotion client:

(1) Check if user is really attempting to promote a file.
    If fs.FileExists(fname) = False Then
      MsgBox "Invalid file specification ... cannot promote folders!", 16
    End If

(2) Asks for destination, QA or PROD
    If sourcelevel = "DEVEL" Then
      Do
pwhere = Wscript.stdIn.ReadLine
pwhere = UCase (pwhere)
Loop Until pwhere = "Q" Or pwhere = "P" Or pwhere = "E"
...
    ElseIf sourcelevel = "QA" Then
      destpath = "PROD\"

(3) The SAS program calling the listener has the following form:
    filename cp socket 'COMPUTER:6001' ;
data _null_;  
file cp;
put '0,SOURCEFILE=<path><name>.SAS';
put '0,FILEEXT=SAS';
put '0,ACTION=C';
put '0,DESTFILE=<path><name>.sas';
put '0,ARCHFILE=<path><name><version>.sas';
put '0,PRMOTER=WAYNE';
put '0,PRDATE=<promotion datetime>';  

The file run tool is scripted similarly with the exception that the extension of the file is checked to determine what utilities are run. For example, "SAS" files will trigger a question to the user on whether to run the program, or compile the program headers of all "SAS" programs in the directory. Files with "LOG" will trigger a question to the user whether to run a log-checking utility.

If UCase(extension) = "LOG" Then
  Do
  runpgm = InputBox("Checklog this .LOG only or all .LOG in directory? This
  .LOG [L], All .LOG [A], Exit [E]? :","File Run Tool")
  runpgm = UCase (runpgm)
  Loop Until runpgm = "L" Or runpgm = "A" Or runpgm = "E"
  If runpgm = "E" Then
    WScript.Quit
  ElseIf runpgm = "L" Then
    run_option = "L1"
  Else
    ...
ADDRESSING KEY SDLC REQUIREMENTS

In this section, we now describe how this approach and architecture addresses the various requirements of a SDLC infrastructure. During this discussion, shortcomings will be identified and ideas for improvement presented.

First of all, standard folder structures are enforced by having processes specifying the use of the New Folder utility to create new directories. Files and documents created by a variety of tools (e.g. SAS, Word, text editors) during the different phases of the lifecycle are placed into these directories.

Beginning with the DEVEL-QA-PROD folder level, the New Folder utility locks down the appropriate directories according to business process requirements. Manual setting of permissions can be performed to open directories, for example, when unblinding a study. The file promotion utility provides the means to transport files between unprotected areas and protected areas. The PROD level is totally read-only to enable safekeeping of production versions of files, versioning, and auditing.

Versioning only occurs at the production level. The reasoning is that when a program has been peer reviewed and finalized, it should be versioned. When working in DEVEL directories, it is the programmer’s responsibility to create backup copies of files as necessary. When a file is promoted into PROD directories, versions are created and records added to the audit dataset describing who promoted the file and when.

Our approach does not contain a true source control element and making it such would require substantial work. There is no check-in and check-out; and it is possible to overwrite a file in the DEVEL area if two or more people work on it at the same time. But this is mitigated by careful assignment of responsibilities and proactive communication. Source control per se was not a priority feature since our programming group is tight-knitted and for the most part, only a single person works in the directory.

However, it probably is not far fetched to add a source control dimension to this infrastructure. For example, file editing can be initiated by using a VBScript tool which first hides or locks the file upon opening it. Or a tool could copy the file to a different location or name when opening the file for edit and setting the read-only attribute of the original file (see paper by Tim Williams).

In our system, change control is achieved by moving the production version back to the development tree, making revisions, and then promoting back upward until the changes once again become a production version.

There also is not a feature to automatically rollback to a prior version. But this can be done manually by examining the archive folder for the desired file and copying it into DEVEL and then promoting it back to PROD to become the newly effective production version.

There may be a few other bells and whistles that are provided by a source control system that we have not implemented. Workarounds are usually available and sometimes delegated to written business procedures.

CONCLUSION

A controlled programming environment requires an infrastructure to support a software development life cycle. The infrastructure can be implemented in a variety of shapes and approaches. Commercial packages exist that costs thousands of dollars but which may be impractical for the variety of SAS programs (e.g. small, one-time) that are written in a pharmaceutical setting. In a Windows environment, using some operating system elements, SAS, and VBScript, we can construct a basic infrastructure that addresses key SDLC requirements.

REFERENCES AND RESOURCES


CONTACT
You comments and questions are welcome.

Wayne Woo
Novartis Vaccines & Diagnostics
Cambridge, MA
E-mail: wayne.woo@novartis.com

SAS and all other SAS Institute Inc. product or service names are registered trademarks or trademarks of SAS Institute Inc. in the USA and other countries. ® indicates USA registration. Other brand and product names are trademarks of their respective companies.