Paper 006-2007

Better SAS® Programming Through Version Control

Tim Williams, PRA International, Charlottesville, VA

ABSTRACT

This paper expands upon concepts introduced in the SUGI31 paper "Version Control on the Cheap. A User-Friendly, Cost-Effective Revision Control System for SAS." Several examples specific to the Concurrent Versions System (CVS) are presented. Those concepts can be applied to other systems. The benefits of version control can be attained by individuals or scaled up to global programming teams.

The goal is not to review and compare various applications and methods, but to present general concepts and provide specific examples from the experience gained using CVS on over three hundred SAS programming projects. With this information you will be well equipped to start research into which tools and methods are best suited to your own programming environment.

Keywords: version control, code management, project management, Concurrent Versions System, CVS, Tortoise CVS, Subversion, open source, coding standards, efficiency

AN INTRODUCTION TO VERSION CONTROL

Informal version control methods are often used in programming without any conscious effort or a dedicated software application. Copies of files from key dates or project milestones may be archived onto backup media or copied into folders named to represent these events. Event name or date identifiers may be added to filenames. These methods of capturing development history rely on labor intensive, error prone methods that result in multiple copies of files.

A version control system captures dates and events as metadata in a central repository. You access only one copy of the project at a time, though each change and event are easily identified and available. Version control provides you with a project "time machine" that allows you to undo changes or return to the state of the project from any previous point in its history. Changes can be easily rolled back if problems occur. If any files are accidentally deleted from the work area they are available from the central repository. Version control also provides security and protection, traceability, audit trails, the ability to implement coding standards, and many more features.

Much of the terminology in this paper refers to the Concurrent Versions System (CVS), but the concepts are more widely applicable. *Version control*, *revision control* and *versioning* all refer to managing the change history of files. In this paper the term revision is used to identify incremental changes to files that are committed to the repository. A *version* refers to a file or group of files at key time points in project development. In CVS a version is identified using a label called a *tag*.

A *repository* acts as a code warehouse where files and their change histories are stored. Repositories provide centralized configuration management, backup, and security with indirect access through an application layer. Programmers *checkout* code from a repository to a local *sandbox* work area for development. Changes are *commited* to the repository for archiving and sharing with the rest of the team. Change history is recorded during each commit and includes the username, timestamp, and comments entered during the commit. A file's complete change history is available for later review and comparison.

Repositories should store the program code needed to recreate all of the project's output. Macro catalogs and format catalogs should be versioned along with SAS program files unless these catalogs are generated from other programs already managed by the system. Other metadata essential to running the programs should also be versioned. Output files and program logs should not be placed in the repository. As a general rule, if a file can be reconstructed from other files already under management, that file should not be versioned. Files generated from source code can easily get out of step when code is changed but not run or when new code is run but the output is not committed.

SAS datasets are an exception to the versioning rule because some systems like CVS do not store binary files efficiently. When these systems are used, binary files should be versioned using other applications.

VERSION CONTROL APPLICATIONS

A review of version control applications is beyond the scope of this paper. Wikipedia is a good starting point for the many reviews and comparisons available online:

http://en.wikipedia.org/wiki/Version_control
http://en.wikipedia.org/wiki/Comparision_of_revision_control_software
http://en.wikipedia.org/wiki/List_of_revision_control_software

Consider your needs, budget, and philosophy during your evaluation. Do you require concurrent development where many people can edit the same file at the same time, or a system where files are locked during editing and unavailable to other programmers? Do you prefer a separate application or would there be greater user acceptance with an interface integrated seamlessly into Windows Explorer? Does the application work well with your SAS environment and coding practices? Can you leverage additional return on investment by using the system with other languages or development environments at your company? These are only a few of the factors to consider while reviewing the many options available. Some of the more popular commercial systems include Visual Source Safe, Perforce and Bit Keeper. Free, open source systems include Concurrent Versions System and a relative newcomer called Subversion.

VERSION CONTROL IN SAS SOFTWARE

Robust version control methods are used in the development of the SAS system. The SAS Institute utilizes customized source management tools based on CVS and maintained by their Tools, Infrastructure, and Process Development Department (SAS Institute, 2006). A degree of version control is available to end users within select SAS applications. SAS Drug Development provides some versioning of code and data. Version control concepts can be applied to SAS/Warehouse Administrator, the SAS Intelligence Architecture and Enterprise Guide. The SAS Institute was invited to contribute to this paper, but was unable to participate before the publication deadline. Please talk with your SAS representative to determine what version control features may be available within your SAS applications.

CONCURRENT VERSIONS SYSTEM (CVS/TCVS)

At PRA we use a combination of applications for version control (Williams, 2006). CVS-NT is the core CVS service. Tortoise CVS (TCVS) provides point-and-click access to CVS commands as drop-down menus within Windows Explorer™. ExamDiff is used to compare changes to files. Both CVS-NT and TCVS are licensed under the GNU General Public License (GNU GPL) and a limited version of ExamDiff is available free of charge. WinMerge and other free open source comparison utilities are also available.

WHAT ABOUT SUBVERSION?

While this paper primarily describes methods for managing SAS code with CVS, a new system called Subversion (SVN) is making rapid gains and is touted as the eventual successor to CVS. It is built on CVS concepts and fixes many of the limitations and performance issues in the older system.

Moving and renaming files and folders is easily accomplished in SVN (not so in CVS). The commit process is much improved under an "atomic" model. If the commit fails part way through the process, the entire commit fails and no changes enter the repository. In CVS, a partial failure during a commit may result in only some changes entering the repository. Though failed commits are rare, their effects can be problematic. In SVN a number of changes can be grouped into a single commit making them easier to track. Better networking support, more efficient labeling, and a database back-end are all features that make SVN worthy of your consideration.

BETTER SAS PROGRAMMING

Many programmers and managers dismiss version control applications as mere archival tools or file stores. They are missing out on a vast array of benefits that can be achieved when version control methods become an integral part of the programming process. When implemented correctly, version control is a platform for enhancing productivity, increasing efficiency, and providing standards for entire teams and companies. The remainder of this paper presents examples of how version control methods can be used to effectively manage SAS program code.

Examples in this paper use CVS-NT and Tortoise CVS, though the concepts may be more widely applied.

SAFETY AND PROTECTION

You may have experienced the agony of making a seemingly minor change to a program only to have the program fail when it is run. What then follows is the laborious task of trying to retract those changes in an effort to return to code that runs without error. CVS takes the guess work out of this process by quickly identifying what has changed

and allowing you to easily roll-back to an earlier revision. Likewise, you can obtain code exactly as it was at key time points using labels (see ***Labeling Files***). If you accidentally delete a program, without version control, it may take a long time to restore it from backup media (assuming you even *have* a backup copy). With CVS you simply right-click on the parent folder and select the *cvs update* command from the menu to retrieve the most recent copy of the file from the repository. Similar protection is provided against losing files by accidentally overwriting them.

Committing changes to a centralized repository provides security and protection. Backup and recovery of the repository fits into your regular server backup cycle.

COMMENT HEADER AND KEYWORDS

Each time a file is committed it can be scanned to determine if the program contains required content. A brief description of these processing scripts is provided in the CVS documentation and Williams (2006). SAS programmers can take advantage of this feature to require that all programs contain a standard comment header. The header may contain versioning information that is automatically updated with each *commit* and *update* event. These CVS keywords identify the name of the program ($RCSfile), revision number ($Revision$), author of the revision ($Author$), and a time stamp of when the revision was committed ($Date$). A complete list of available keywords can be found in most CVS documentation (Cederqvist, 2005; Thomas and Hunt, 2003; Vesperman, 2003).

Here is an example comment header before addition to CVS:

```
/*************************************************
 - - - - - DO NOT EDIT THE NEXT 4 LINES  - - - - - - -
PROGRAM NAME   : $RCSfile$
REV/REV AUTH   : $Revision$ $Author$
REV DATE       : $Date$  UTC
 - - - - - DO NOT EDIT THE 4 LINES ABOVE - - - - - - -
CLIENT/PROJECT: ACME Products/Anvil 2000
PURPOSE        : Sales summary for the Anvil 2000 model
ORIG AUTHOR    : Tim Williams
  ... etc.
*************************************************/
```

After the first commit of the file the keywords resolve to their values. Here you see that the file, QuarterlyReport.sas, was committed as revision 1.1 by Tim Williams on 20 January, 2006.

```
/*************************************************
- - - - - DO NOT EDIT THE NEXT 4 LINES  - - - - - - -
PROGRAM NAME   : $RCSfile: QuarterlyReport.sas,v $
REV/REV AUTH   : $Revision:1.1 $ $Author: WilliamsTim $
REV DATE       : $Date: 2006/01/20 16:27:11 $  UTC
 - - - - - DO NOT EDIT THE 4 LINES ABOVE - - - - - - -
CLIENT/PROJECT: ACME Products/Anvil 2000
PURPOSE        : Sales summary for the Anvil 2000 model
ORIG AUTHOR    : Tim Williams
  ... etc.
*************************************************/
```

The revision number increments automatically with each commit of the file. The number can be used as a link to quality control checklists and replaces error-prone methods of manually entering identifiers or relying on file system dates.

FILE NAMING AND CONTENT

Because files are scanned during commits, it is possible to enforce additional restrictions on file content beyond comment headers and CVS keywords. You may choose to require that all `libname` definitions be placed in a particular file. Format definitions are another good candidate for centralization in a single file. Then, if `libnames` or format definitions appear in files other than the approved file names, the commit will fail with a warming message to the programmer that the content should be relocated. After the content is moved to the correct files the commit will succeed. Similar restrictions can be placed on file names to ensure they follow a naming convention.

Forcing these restrictions upon programmers is a very heavy-handed approach and one that can greatly compromise user acceptance. We have not implemented these restrictions at PRA. Instead, we rely on coding guidelines for file naming and content. We do restrict commits to .SAS filename extensions and a few exceptions.

FOLDER STRUCTURE AND NAMING

One of the easiest ways to provide standardization is during project set up. The process is invisible to the programmer and offers a consistent folder structure across all projects.

A project template is set up as its own module in the CVS repository, complete with standard macros and code templates. Administrators maintain the standard template and update it with new content as it becomes available. When development is ready to start on a new project, an administrator executes a Perl script that prompts for the new project identifier and location of the CVS repository (we use multiple CVS repositories at PRA). The script checks out the project template and replaces the template identifier with the new project name. The project is then added and committed to the appropriate repository using the new name. When the process is complete, an email is automatically sent to the project mail list announcing that the project is ready for development activity. The entire set up process can be completed in under a minute.

SHARED CODE BASE

Sharing code between projects or within an entire company can be better managed using version control. Several approaches are possible. I will illustrate one possible method for a company-wide validated macro catalog and a separate method for sharing code in a subset of related projects.

Maintaining code for a company-wide validated macro catalog may be best managed as its own project by a select group of skilled programmers. These programmers are responsible for adding new code and releasing the compiled catalog after comprehensive evaluation. After code evaluation is complete, the compiled catalog is added to the standard project template and automatically becomes part of any new project set up from that date forward. The new catalog is also posted on the company intranet and programmers are informed of the changes by email. Older projects are updated by the lead programmer downloading the catalog from the intranet and committing it as a replacement to the older catalog in the project. The release and revision information for the catalog contents are easily determined by the project team when coding guidelines are followed (see ***Managing Validated Macros***.)

There are potential pitfalls when multiple projects rely on a shared code base. I caution against providing this code in a shared location for use in a large number of loosely related projects. A change required for one project may unexpectedly break code or invalidate output in other projects.

Another approach is possible when a small number of similar projects share code. In this example three projects for a company named "DrugCo" share SAS macro code. A sandbox folder structure is set up with the company name as a parent folder and each project below it (**Figure 1**: Project1, Project2, Project3). At the same level as the projects is another folder named "DrugCo-SHARED". This folder contains code that is managed as a separate module in CVS. It is referenced by all of the DrugCo studies using `filerefs` that point to the /SAS/Macros/ folders in the DrugCo-SHARED structure.

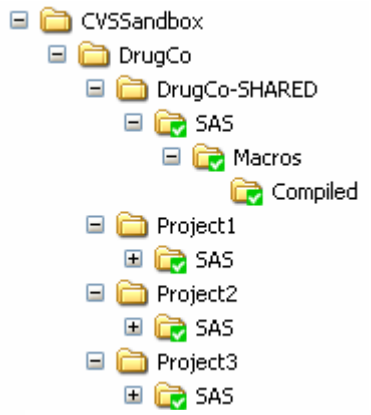


Figure 1: Shared Code Module.

If code in the DrugCo-SHARED module requires customization that is unique to one of the projects, that code should be moved out of the shared location for management within each project.

CVS modules can also be linked to automatically checkout a shared module when an individual project is requested. In this situation, a checkout of Project1 would also result in the DrugCo-SHARED module alongside the Project1 structure. This is an advanced method beyond the scope of this paper. It requires a very clear understanding of how the shared module interacts with all of its associated projects. The linked method is best left to small projects worked on by advanced CVS users.

SEPARATION OF DEVELOPMENT AND PRODUCTION ENVIRONMENTS

Good programming practices recommend development and debugging of code in an area separate from where validated code is run to produce the final output. This can mean manually moving programs from a development area to a production area after the programs have passed a quality assessment. This approach results in copies of programs in multiple locations as production runs are completed and archived during the project life cycle.

Use of CVS avoids uncontrolled copies of files. Programmers label file and project events using *tags*, without the need for copies in multiple locations on the file system. Development and production areas are in separate locations on the file system. A programmer labels the project with a production tag, then uses that label to checkout program code into the production area for execution. Old log files and programs that are not labeled are left behind in the development area. In this way, the production area has a clean starting point that contains only the appropriate files.

CVS lacks a code promotion facility where programs move from development through quality control and finally into production. Tags and a strong methodology overcome this limitation and help programmers manage their code through the project lifecycle.

CODE PORTABILITY

Coding `libnames`, `filerefs`, and other path information in multiple files greatly increases the maintenance burden when files are moved to a new location. A first step toward simplification is coding paths into global macro variables within in a single file that is executed first within the SAS environment. Ideally, paths can be set dynamically at run time. This is important for developers that supply program code to clients who use a different folder structure. When paths are set dynamically there is no need to alter programs when code is shipped to the client or when validated programs move from a development to a production location. It also allows concurrent development in multiple, independent sandboxes. This is a key requirement for SAS programs managed using CVS.

There are many approaches to obtaining path information and assigning it to a macro variable. One solution is to launch SAS using a Visual Basic script that is managed as part of the project. Each project contains this script. It is used to launch the SAS Display Manager system using SASOACT.EXE as described by Hunt (2005). The script also assigns the path to the project's SAS folder to a global macro variable named SASPath. As standard practice, we store a SAS program with all libname assignments in the same folder as the VB script. The script writes the path information into the value of -SASINITIALFOLDER in the config file and then opens the Enhanced Editor. Setting the value of project path in -SASINITIALFOLDER allows efficient file open and save actions by the programmer.

Libnames and other path references can be set dynamically when the path is available in the local SAS session:

```
libname  macLib "&SASPath\SAS\Macros";
libname macStan "&SASPath\SAS\Macros\Standard";
```

The VB script also detects if SAS was launched from a development or production environment and writes a corresponding message to the SAS log file.

Restrictions in your local computing environment may limit code portability. If your development sandbox is located on your workstation and you use SAS/CONNECT to remote submit programs to a SAS server, that server must be able to reach back onto your workstation to obtain macro code or other programs. This is only possible when your company permits mapped workstation drives or other access methods from servers to workstations. If this is not allowed, your code must be moved from the workstation environment to a file server that can be accessed by the SAS server.

CONCURRENT DEVELOPMENT

Version control allows a team of programmers to work on a project without interfering with one another. Programmers may sit in multiple offices throughout the world or take work home with them without fear of having their code overwritten by others. By default, there is no "file locking" in CVS, so programmers can edit the same files simultaneously. Everyone's changes are merged together in the repository when the file is committed.

In the concurrent development model no one can lock a file and then go home or on vacation and make that file unavailable to the team. Communication becomes critical because the system does not automatically tell you who is working on which files. Only when two programmers both commit their changes do they find out that they were working on the same section of the same program. This can lead to conflicting changes, but CVS is very good at detecting and resolving code conflicts. It is best to avoid potential conflicts by assigning files to individuals. Tools like CVS Mailer for CVSNT assist team communication by sending email notifications for commit and tag events to the entire team. Without this type of notification, there is no way to know that a new revision has been committed until you update your own work area or commit the file.

PARALLEL DEVELOPMENT

Development can be split onto separate development “timelines.” There is no need to stop moving toward your next code release when making changes or corrections to past work. In our SAS programming environment this is a relatively rare occurrence. It is useful in situations where you want to test a significant change without fear of disrupting other programming. The testing can be isolated onto a development *branch*, tested, then merged back into the development "mainline" after the code has passed evaluation. The section ***Labeling Files*** provides an example of how corrections to past work can be isolated onto a branch and merged back into the main development line.

FILE AND PROJECT STATUS

TCVS displays the status of each file in a sandbox using icon overlays within Windows Explorer. A file that has been changed and requires a commit is easily differentiated from those that have not been changed or are not managed by CVS.

The *cvs status* command summarizes the status of all files in a sandbox, but does not provide a concise summary. A SAS macro can parse the output from this command to display an easily read overview in the SAS output window using ODS HTML (**Appendix 1**). The summary can be accessed from the click of a button when the macro call is attached to a toolbar in the Enhanced Editor. This summary is preferred for obtaining an overview of the entire sandbox. Details for individual files are best obtained using the revision graph (**Figure 2**) or the CVS History window in TCVS.

Additional tools are available to help navigate and summarize repositories. ViewCVS provides a browsable graphical representation of the repository. CVS Monitor is another repository browser that provides graphical summaries of repository activity and illustrates which projects and programmers are the most active. CVS Monitor is available only for Linux/Unix systems.

LABELING FILES

CVS allows you to label a file, group of files, or an entire project. The label can later be used to recreate the files exactly as they were when the label was applied. Labels aid in identifying what has changed between releases of a project and can be used to move code from a development to a production environment.

Tags and *branches* are two types of labels available in CVS. At PRA we use tags to identify key time points in the project lifecycle. The more complex branch label is used is isolate a previous set of code for changes or corrections.

Developing standards for label names is vital to project success. Work with your teams to develop a list of events that must be identified during the project lifecycle. You may then restrict the labels to an approved list for consistency and standardization (Williams, 2006).

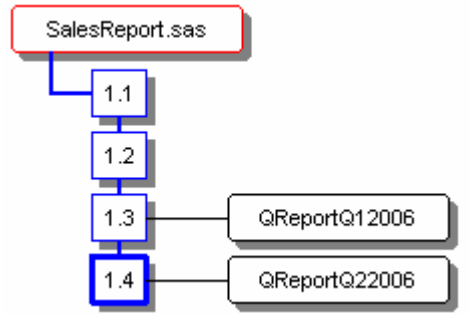
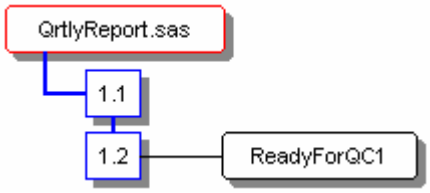
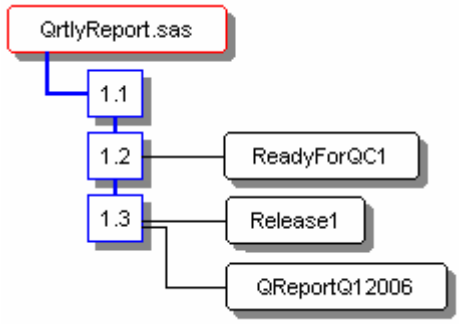
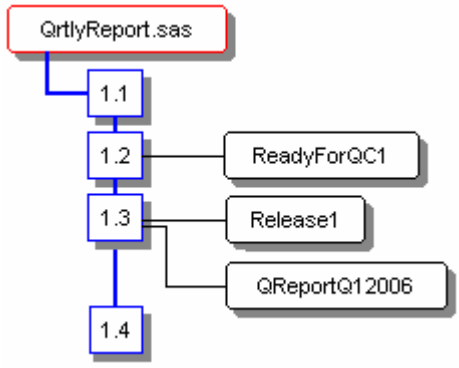
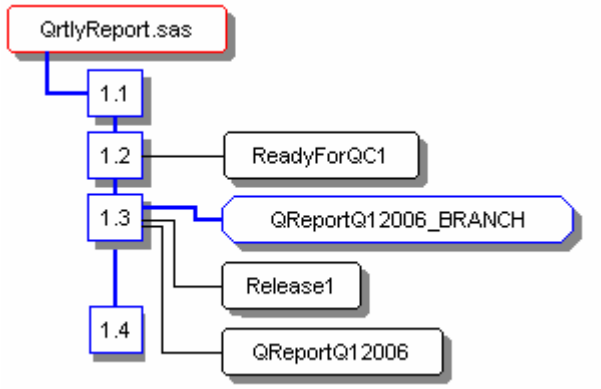


Figure 2: Tags on revisions in the TCVS Revision Graph).

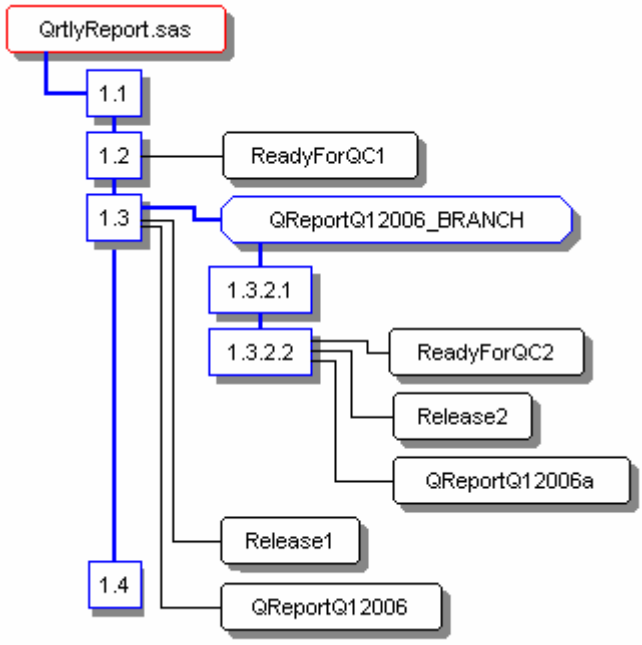
Of equal importance is the methodology for how the labels are used. Following a simple methodology will ensure compliance and user acceptance. A complex system of branching and tagging is confusing and will not be followed by programmers, thus defeating the advantages of their use. Tagging and branching concepts are best explained by way of an example. The example illustrates one possible approach. Your own implementation would undoubtedly vary. A single file is followed through its development lifecycle. All programs in the project progress along similar parallel paths.

All figures in this section are screen shots from the Tortoise CVS revision graph.

A programmer writes code that will produce a quarterly report of sales figures. The file is committed to the repository as revision **1.1**. More changes are needed and the file is committed a second time as revision **1.2**. It is then labeled with the tag **ReadyForQC1** to indicate the file is ready for its first quality control evaluation.	 **3a**

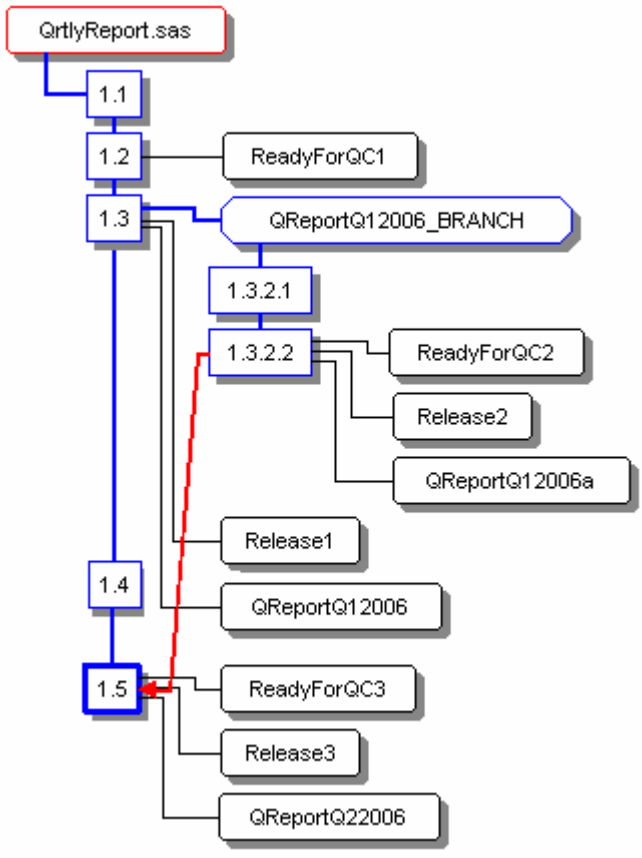
During the QC process a minor change is made and committed (**1.3**). The file is then tagged **Release1** - the first time the file is ready for a production release. Some time later the entire project is tagged **QReportQ12006** for the First Quarterly Report in 2006. The project is moved into production (not shown) and run to create the report.	 **3b**
Programming continues toward the next quarterly report. Calculations are added for sales in the second quarter and the program is committed as revision **1.4**.	 **3c**
An error is found in the first quarter sales calculations. Rather than lose all of the code already written for the next report, the programmers go back and make a correction to the first report by ***branching*** revision **1.3** onto a second development line using the branch name **QReportQ12006_BRANCH**.	 **3d**

Changes are made on the branch and committed as revisions **1.3.2.1** and **1.3.2.2**. Revision **1.3.2.2** is tagged as ready for evaluation (**ReadyForQC2**) and later for release (**Release2**). All files are labeled with the tag **QReportQ12006a** when the project is ready for the corrected run of the first quarterly report. The ***a*** in the tag name is used to identify that version as the first *correction* to the QReportQ12006 production run. If a later correction was needed it would be labeled QReportQ12006**b**.



3e

The corrections for the first quarterly report must appear in all later reports. These changes are easily merged back into the development mainline and committed as revision **1.5**. This revision goes through the evaluation (**ReadyForQc3**) and release (**Release3**) steps before labeling for the second quarterly report (**QReportQ22006**) .



3f

TRACEABILITY AND CODE HISTORY

When you commit changes to the repository your username is associated with the new revision as well as each line you changed within the file. During the commit process you may enter a description of your changes. There is no need to list the exact changes in your comment because they are best viewed by comparing revisions in a process termed "diffing." Applications like ExamDiff, WinMerge, or UltraCompare provide a graphical display of changes between two revisions.

Comments entered during a commit can be viewed in the CVS History or CVS Revision Graph. Passing the mouse over the revision numbers in the graph reveals the time stamps, usernames, and commit comments. The graph is not solely for display purposes. You can select any two revisions and send them into the comparison application, retrieve any revision into the sandbox, or apply labels.

The *cvs annotate* facility provides a line-by-line change history for a file and shows who last altered each line of code. This audit trail is particularly helpful if you want to contact the programmer to determine why a specific change was made. Annotate shows the following information for each line:

- Revision when the line was last altered. Lines marked as 1.1 have not changed since the first revision was committed.
- Who last edited the line.
- When the line was changed.
- Current line content.

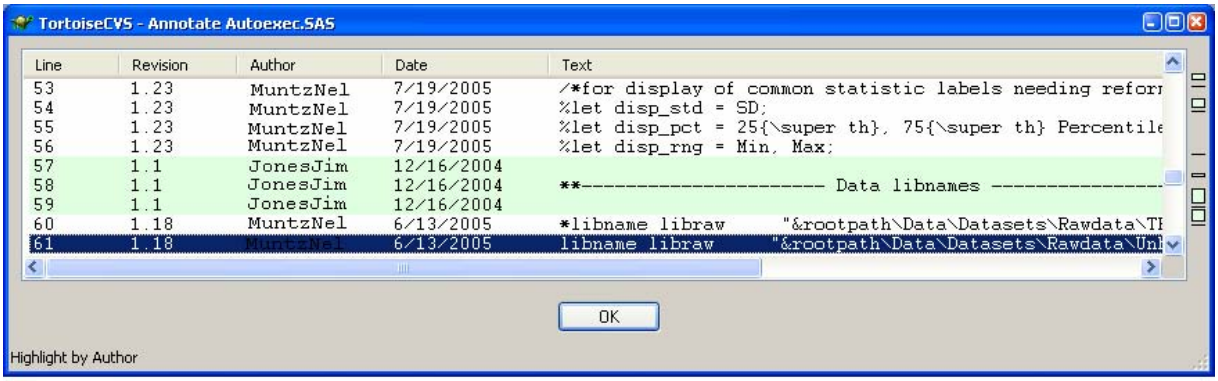


Figure 4: CVS Annotate Displayed in TCVS.

MANAGING VALIDATED MACROS

Version control can play a key role in the distribution of compiled macro catalogs and identification of their constituent macros. When a catalog is part of a project template in the repository it can be distributed as part of the set up process. The project template is updated with the latest versions as macros are changed and re-validated. Programming teams can update existing projects with the newest macros released on the company intranet or other centralized location. Catalogs could be stored in a shared location on a file server, but this approach is not recommended (see ***Shared Code Base***.)

Macro catalogs can be compiled from source code or you may wish to distribute a compiled catalog that is not available for alteration by programming teams. This allows distribution of validated macros that can not be altered. Requests for changes to the macros can be coordinated by a central development team.

CVS revision numbers can be placed in PUT statements and the macro description parameter (`/DES`) to identify the revision of the macro present within the compiled catalog. The catalog itself can be versioned and identified using a global macro variable that is added during the catalog's build process. In the example code below we build a catalog that contains two macros. **_release** is a utility macro whose sole purpose is to display the catalog release value in SAS program logs. **_logChk** is a macro that parses log files looking for errors, warnings, and other problems. SAS logs should identify which version of the compiled catalog is in use and which revision of _logChk is compiled within the catalog.

The catalog release version is identified using the macro variable **_catRel**. The value of _catRel should match the tag applied to the catalog when it is compiled and released. Note how the $Revision$ keyword is used in both the /DES option and PUT statements in the macro source code. Its value is updated automatically each time a new revision is committed to the repository.

_release Macro Source Code (_release.sas)

The value of _catRel is hard coded into this file and should match the value of the CVS tag that will be applied for the next release. Some forethought is required because the value is hard coded *before* the file is committed. The matching tag is added after the catalog is compiled and committed as a new revision. The value for the revision number is not edited by the programmer. In the example code below the number 1.6 in both the /DES and %PUT statements will automatically increment to 1.7 the next time the file is committed. The value of _catRel is set to Release-1-4-0, the same tag that will be applied after the catalog is committed.

```
** Sets the catalog release value. Value should match the next CVS tag **;
%MACRO _release /DES="Set the compiled catalog release value.
                     $Revision: 1.6 $ " store;
  %global _catRel;
  %PUT Utility macro _release $Revision: 1.6 $ ;
  %LET _catRel=Release-1-4-0;  ** Must match CVS tag **;

%MEND _release;
```

_logChk macro source code (_logChk.sas)

In this excerpt of the macro code the $Revision$ keyword is placed in %PUT statements to identify the revision in log files. The value of $Revision$ is updated automatically with each commit. The call to the _release macro sets the value of _catRel which is also displayed in the log using %PUT statements.

```
%macro _logChk(_srcDir=, _fileChk=all, _outFile=LogReviewSummary.txt)
               /DES="Log Check Macro. Parse log files for ERROR, WARNING
                 and specific NOTE messages. $Revision: 1.37 $" store;
  OPTIONS nosource nosource2 nomprint nomlogic nosymbolgen;
  %_release;  ** Obtain Catalog release value **;
  %PUT;
  %PUT +-------------------------------------------------------------------+;
  %PUT |;
  %PUT |                    Macro Catalog &_catRel;
  %PUT |;
  %PUT |                     _logChk $Revision: 1.37 $ ;
  %PUT |                Review LOG files for significant messages;
  %PUT |              ** Is not a substite for manual QC log reviews **;
  %PUT +-------------------------------------------------------------------+;
  %PUT;
  OPTIONS source source2;
```

…. remainder of macro code omitted…

Build file (buildCatalog.sas)

Both macros are added to the catalog using a build file that specifies the location where the macro catalog should be saved. The program then %INCLUDEs the constituent macros (_release, _logChk).

```
LIBNAME macLib "C:\foo\macros\compiled";
OPTIONS mstored sasmstore=macLib;
%INCLUDE "C:\foo\macros\source\_release.sas";
%INCLUDE "C:\foo\macros\source\_logChk.sas";
```

After the build file is run, the resulting SASMACR.SAS7BCAT catalog file is committed to the repository and tagged with the label "Release-1-4-0" to match the value of _catRel (as assigned in the _release.sas program). The compiled catalog is then copied to the project setup template for use in all new projects. The catalog is also advertised on the company intranet where programmers may obtain the file to update their existing studies. When the catalog is called, the log will display information that clearly identifies the release of the catalog and the revision of the macro that was called within it:

```
+-----------------------------------------------------------------+
|                    Macro Catalog Release-1-4-0
|
|                     _logChk $Revision: 1.37 $
|                Review LOG files for significant messages
+-----------------------------------------------------------------+
```

Because the $Revision$ keyword was also used in the /DES parameter, revision numbers are available when using PROC CATALOG to easily identify the revision of each macro within the compiled catalog.

LIMITATIONS

Use of version control does not imply that the code is in any way superior to code which is not versioned. Version control provides an audit trail of code development and it makes it easier to identify revisions and link them to quality control methods. The code itself is still only as good as the programmers that write and evaluate it. Versioning must be part of a larger process that includes programming standards and validation methodologies.

CVS has some technical limitations that warrant discussion. Renaming files is difficult, though it is becoming easier in recent releases of CVS. Renaming or deletion of folders (directories) is also problematic, but these events are rare in a well managed programming environment.

Branch and tag names can not be time stamped or altered automatically by the system. These labels can only be evaluated in a pass/fail context to determine if the labels fit a naming convention. Unless additional scripts are used, it is not possible to determine who applied a label and when it was attached to a revision.

TCVS provides a convenient user interface for most users; however, administrators will find themselves relying on the CVS command line for more advanced tasks. Although WinCVS provides a more comprehensive feature set, it lacks the simple and efficient integration with Windows Explorer.

CVS is not integrated with an electronic signature system, so the change history for program code is a rudimentary audit trail at best. CVS is not a good environment for storing SAS data sets and other binary files. You may wish to consider applications such as SAS Drug Development® for additional security features and the ability to manage code and data together in the same application.

LESSONS LEARNED

Many lessons were learned in the past three years while implementing CVS/TCVS on over three hundred projects. The first and most important is to provide adequate training to all programmers. Instruction for new programmers and refresher courses for existing staff are critical to success. Obtaining feedback from staff is a great way to improve the system. Do not rely on email or formal meetings for this information. Sit down with programmers for one-on-one conversations to hear their concerns and suggestions.

Keep the code management process simple and consistent. Use tags to identify key events and avoid complex branching and merging. The code conflicts sometimes encountered during merges and updates can be confusing, in part because they are rarely encountered.

Appoint more than one version control administrator who is intimately familiar with the entire system. Administrators must be able to address issues that range from complex background scripts, to project set up, to troubleshooting. Do not be draconian in your standards and restrictions. As an administrator, your goal is to provide a tool that works efficiently within the SAS programming environment. Work with your programmers to develop the system and do not force your code development ideals on the team. Develop standards incrementally with programmer guidance. Do you really want to force libname and format definitions into a specified SAS program?

Programming in a concurrent development environment can be a large paradigm shift for SAS programmers. Code

must be committed often and sandboxes must be updated to receive changes committed by other programmers. If specific files are used to store formats and path definitions, it is best to designate one or two programmers to edit and commit those files; rather than a large number of programmers all adding content concurrently. This latter situation can lead to "false conflicts" when updating and committing files.

Finally, *free* software does have costs associated with it. Before you inform senior management that you are about to implement a solution that "won't cost a dime!" consider the costs of validation, hardware, implementation, training, and support (Williams,2006).

CONCLUSION

Version control can become a seamless and enjoyable part of your programming experience. Or it can be cumbersome, inefficient, frustrating, and soon abandoned. A well-planned implementation of the proper version control solution will have you wondering how you ever programmed without it.

APPENDIX 1

EXCERPT FROM A SANDBOX SUMMARY GENERATED USING A SAS ODS MACRO

C:\CVSSandbox\DrugCo\Project1\SAS
Last updated: 17JUL2006:15:30:51
Files Summarized: 22

Path	File Name	Sbx Revision	Sbx Status	Tag	Tag on Revision
/SAS	QrtlyReport.SAS	1.4	Needs Update		1.1
				ReadyForQC1	1.2
				Release1	1.3
				QReportQ12006	1.3
				QReportQ12006_BRANCH	1.3
					1.3.2.1
				ReadyForQC2	1.3.2.2
				Release2	1.3.2.2
				QReportQ12006a	1.3.2.2
					1.4
				ReadyForQC3	1.5
				Release3	1.5
				QReportQ22006	1.5
/SAS/Reports	SalesReport.SAS	1.4	UP-TO-DATE		1.1
					1.2
				QReportQ12006	1.3
				QReportQ22006	1.4
	Profit.SAS	1.1	**LOCALLY MODIFIED**		1.1

Status of Needs Update indicates other programmers have committed changes: use CVS Update to obtain the latest changes.

NOTE: Colour coding for Sandbox Status (Sbx Status) and special tag names will not be evident when printed in black and white.

REFERENCES

Cederqvist, Per et al. 2005. *Version Management with CVS.* Free Software Foundation, Inc. Available in stable release and WIKI formats. <http://ximbiot.com/cvs/manual/> (February 26, 2007).

CVS-NT. <http://www.march-hare.com/cvspro/> (February 26, 2007).

Fogel, Karl. 2000. *Open Source Development with CVS*. Published under the GNU GPL by the Free Software Foundation and available at: <http://cvsbook.red-bean.com/> (February 26, 2007).

Free Software Foundation. 1991. *GNU General Public License, Version 2*. <http://www.gnu.org/licenses/licenses.html#GPL> (February 26, 2007).

Hunt, Stephen, Tracy Sherman and Brian Fairfield-Carter. 2005. "An Introduction to SAS Applications and the Windows Scripting Host." *Proceedings of the Thirtieth SAS Users Group International Conference*, Philadelphia, PA, Paper 226-30.

Liebman, Roberto. 2003. Comments in the Slashdot Book Reviews forum for *Pragmatic Version Control Using CVS*. <http://www.slashdot.org> (February 26, 2007).

March Hare Pty Ltd. Commercial support for CVSNT. <http://www.march-hare.com/index.htm> (February 26, 2007).

Mason, Mike. 2006. Pragmatic Version Control Using Subversion. 2nd Edition. The Pragmatic Programmers LLC, Raleigh, NC.

Purdy, Gregor. 2000. *CVS Pocket Reference*. O'Reilly & Associates, Inc, Sebastopol, CA.

SAS Institute Inc. 2006. *"The Quality Imperative - SAS' Commitment to Quality*. <http://www.sas.com/whitepapers/index.html> (February 26, 2007).

Subversion. <http://subversion.tigris.org/> /> (February 26, 2007).

Thomas, David and Hunt, Andrew. 2003. *Pragmatic Version Control Using CVS*. The Pragmatic Programmers LLC, Raleigh, NC. <http://www.pragmaticprogrammer.com> (February 26, 2007).

Vesperman, Jennifer. 2003. *Essential CVS*. O'Reilly & Associates, Inc, Sebastopol, CA.

ACKNOWLEDGMENTS

I wish to thank the SAS programmers at PRA International for their ongoing use and support of open source version control.

CONTACT INFORMATION

Your comments and questions are valued and encouraged. The author may be contacted at:

Tim Williams
SAS Systems Administrator
PRA International
4105 Lewis and Clark Drive
Charlottesville, VA 22911

Email: WilliamsTim@PRAIntl.com
SASSysAdmin@gmail.com