Paper SAS1890-2015

Someone Changed My SAS® Visual Analytics Report!

How an Automated Version Control Process Can Rescue Your Report and Save Your Sanity

Jerry Hosking, SAS Institute Inc.

ABSTRACT

Your enterprise SAS® Visual Analytics implementation is on its way to being adopted throughout your organization, unleashing the production of critical business content by business analysts, data scientists, and decision makers from many business units. This content is relied upon to inform decisions and provide insight into the results of those decisions. With the development of SAS Visual Analytics content decentralized into the hands of business users, the use of automated version control is essential to providing protection and recovery in the event of inadvertent changes to that content. Re-creation of complex report objects accidentally modified by a business user is time-consuming and can be eliminated by maintaining a version control repository of report (and other) objects created in SAS Visual Analytics. This paper walks through the steps for implementing an automated process for version control using SAS®. This process can be applied to all types of metadata objects used in multiple SAS application development and analysis environments, such as reports and explorations from SAS Visual Analytics, and jobs, tables, and libraries from SAS® Data Integration Studio. Basic concepts for the process, as well as specific techniques used for our implementation are included. So eliminate the risk of content loss for your business users and the burden of manual version control for your applications developers. Your IT shop will enjoy time savings and greater reliability.

INTRODUCTION

This paper provides a step-by-step approach to implementing an automated version control process for metadata objects such as reports and explorations created in your SAS® Visual Analytics environment. For us, the Enterprise Solutions Data and Analysis team (ESDA) in the SAS IT division, this is essential to provide protection and recoverability for the content being produced in our enterprise SAS Visual Analytics implementation. However, the methods for the automated version control process can be easily generalized to apply also to the wide variety of metadata objects within our entire high-performance analytics and enterprise business intelligence (EBI) environments, as well as our clients' SAS Visual Analytics content.

ESDA supports multi-tiered enterprise SAS® High-Performance Analytics, SAS Visual Analytics, and EBI environments and their associated clients. These environments house our enterprise data warehouse and host our high-performance business analytics to meet the business decision making and forecasting needs of SAS. These are the primary clients interfacing with the HPA and EBI environments:

- SAS Visual Analytics – for data exploration, providing a client interface to data, analysis and reporting
- SAS® Data Integration Studio – to develop code for automated and ad hoc computing
- SAS® Management Console – to manage all the components of our environments
- SAS® Enterprise Guide® – for data exploration and testing
- Platform Process Manager – to implement automation and scheduling of job execution

Most of these interfaces produce metadata objects that contain the elements from which we build our business applications. Because the version control process is the same for metadata objects produced within SAS Visual Analytics as the other clients, we have applied this process broadly throughout all tiers of both environments.

In this paper, references to metadata objects can be interpreted as a report or exploration in SAS Visual Analytics or a job, table or library in SAS Data Integration Studio, for example. It is the stored metadata

that defines the content of a report and its connection to data, or the columns in a table and its physical location.

Following the steps outlined here, you can implement a version control process to protect the metadata of your SAS environment. The process described here differs from the full metadata server backup and restore process that is part of normal system administration (as described in the SAS® 9.4 Intelligence Platform: System Administration Guide) in that this process operates at the individual metadata object (an individual report, for example) level. That means you have specific control over the backup and restore of an individual piece of content or object.

PROCESS STEPS

Figure 1 shows the basic steps in automating the version control process:

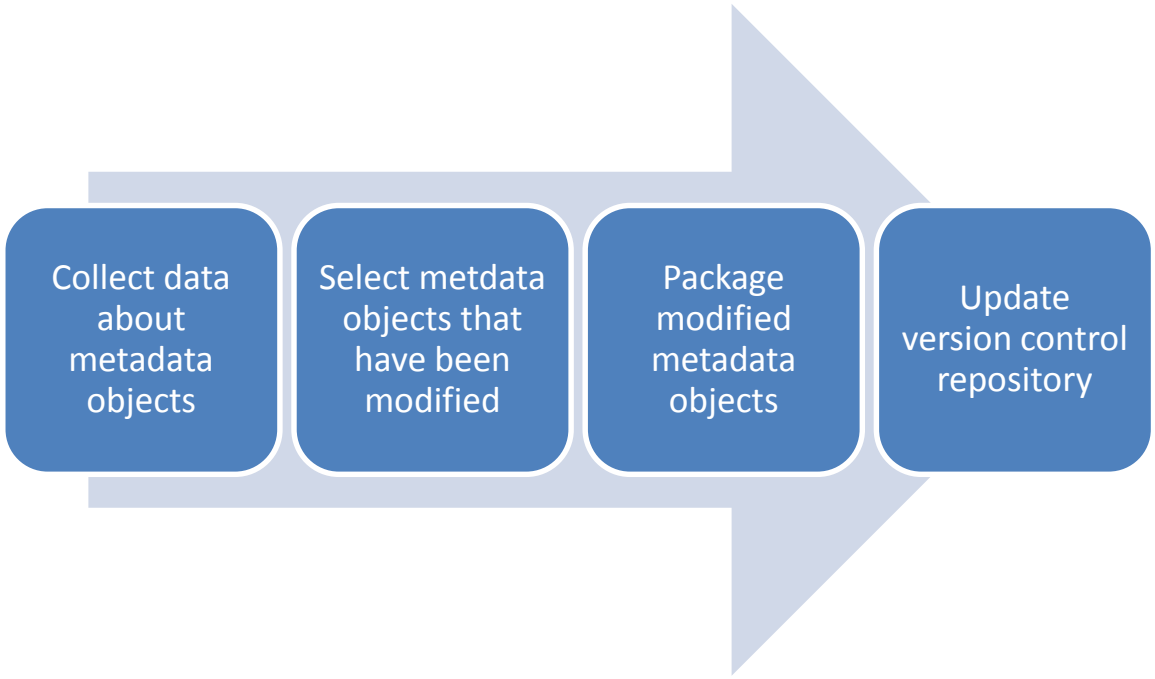


Figure 1. Basic Steps for Automated Version Control Process

Within those four basic steps, there are multiple tasks built to provide robust functionality and assure processing reliability, such as log management and error checking. Those tasks will be enumerated and described in the following sections.

STEP 1: COLLECT DATA ABOUT METADATA OBJECTS

Figure 2 lists the major tasks associated with collecting data about your metadata objects.

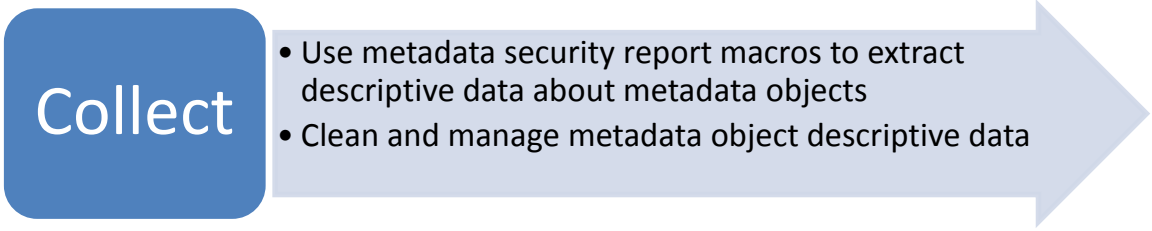


Figure 2. Metadata Collection Tasks

Metadata Security Report Macros

The metadata security report macros, as documented in SAS 9.4 Intelligence Platform: Security Administration Guide, are designed to help report on and administer security for the metadata objects in your environment. To provide reporting, these macros collect all the identifying and descriptive data, as well as the security and permissions data, for each metadata object. These macros are typically located in the SAS Foundation SASAUTOS location ($SASROOT/sasautos) of your SAS installation. Run the macros, directed at your metadata repository, in order to collect the descriptive data about your metadata objects. This descriptive data is the source data needed to build the code that processes the metadata objects for archiving.

Task 1: Connect to the Metadata Server

Unless your SAS session running the security macros has a connection to your metadata server, run code to establish a connection to the metadata server that contains your metadata repository as shown in Code Sample 1:

```
/* connect to the metadata server */
options
metaserver=machine-name
metauser="sasadm&saspw"
    metapass="{SAS002}3CD4EA1E35CA49324A0C4D63";
```

Code Sample 1. Connect to the Metadata Server

For your connection to the metadata server, use a username that has sufficient authorization to access and examine all of the metadata objects that you plan to include in your version control process.

Task 2: Invoke the Metadata Security Report Macros

The simplest way to run the macros is to specify only the FOLDER= parameter and allow the entire %MDSECDS macro and its underlying macros to run as shown in Code Sample 2:

```
/* run the main report macro against a target folder */
%mdsecds(folder="\demo");
run;
```

Code Sample 2. Invoke the Metadata Security Report Macros

The folder specification begins with the highest level folder location in your metadata repository and traverses to the folder location in which you want to start your data collection. Figure 3 shows a sample directory structure:

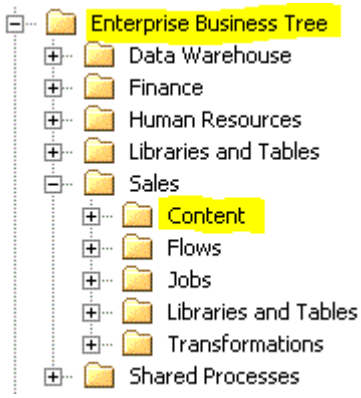


Figure 3. Sample Directory Structure

To execute version control on the **Content** folder that is subordinate to the **Sales** folder, the FOLDER= macro parameter value is as follows:

folder="\Enterprise Business Tree\Sales\Content"

To execute version control on the entire tree, the FOLDER= macro parameter value is as follows:

folder="\Enterprise Business Tree"

Running the %MDSECDS macro produces five work tables. The table of interest is MDSECDS_OBJS. This table contains the name and many descriptive characteristics for each metadata object in the folder tree specified in the macro invocation.

Consider these two additional options for using the %MDSECDS macro:

1. Use a parameter to write the output to a permanent table.

   If you are going to be collecting metadata information from only one folder tree, write the output directly to a permanent table. To do that, issue an appropriate LIBNAME statement that points to the permanent location and then add the OUTDATA= parameter to the macro invocation:

   OUTDATA=permloc.MDSECDS

   If you are collecting metadata information for multiple folder structures, write to differently named locations in the WORK library and then combine the resulting temporary tables into one permanent table:

   OUTDATA=work.SALESJOBS

   The advantage of this option is the creation of a permanent table containing useful information about your metadata objects. It's surprising how many times it has been handy to have such a table since we started keeping it.

2. Make a customized version of the MDSECDS macro to eliminate unneeded processing.

   The %MDSECDS macro calls multiple underlying macros. For the purposes of collecting the metadata needed for version control, it is not necessary to run all of the underlying macros. Comment out the call to %MDSECGP, %MDSECTR, and %MDSECVW macros to eliminate running the unneeded macros. Remember not to change the original version in your installation. Either make a copy in your autocall library (with a different name) or include the macro code in your job with the appropriate sections removed. The advantage of this change is that the job executes quicker and extraneous tables are not created. The disadvantage is that if changes are made to the original metadata security macros for new software versions, you do not automatically pick up those changes.

   Output 1 shows an excerpt from the PRINT procedure for the resulting MDSECDS_OBJS table produced by the %MDSECDS macros:

Sample Excerpt from MDSECDS_OBJS

Obs	ObjName	Location	PublicType	MetadataUpdated
1	Extract_Customer_Data	/Enterprise Business Tree/Sales/Jobs/Extract/	Job	13Oct2014:15:04:22
2	Extract_Product_Data	/Enterprise Business Tree/Sales/Jobs/Extract/	Job	13Oct2014:16:06:49
3	Extract_Sales_Data	/Enterprise Business Tree/Sales/Jobs/Extract/	Job	14Oct2014:15:05:04
4	Load_Master_Customer_Data	/Enterprise Business Tree/Sales/Jobs/Load/	Job	16Oct2014:15:06:12
5	Load_Product_Sales_Data	/Enterprise Business Tree/Sales/Jobs/Load/	Job	16Oct2014:15:06:43
6	Combine_Product_Sales_Data	/Enterprise Business Tree/Sales/Jobs/Transform/	Job	20Oct2014:15:07:08
7	Filter_Customer_Data	/Enterprise Business Tree/Sales/Jobs/Transform/	Job	20Oct2014:15:16:18
8	Archive_Discontinued_Products	/Enterprise Business Tree/Sales/Jobs/Utilities/	Job	21Oct2014:15:16:42
9	Staff Assignment Plans	/Enterprise Business Tree/Sales/Content/Management/	Report.BI	22Oct2014:14:52:48
10	Staff Adjustments - Additions	/Enterprise Business Tree/Sales/Content/Management/	VisualExploration	22Oct2014:14:54:35
11	Staff Adjustments - Transfers	/Enterprise Business Tree/Sales/Content/Management/	VisualExploration	22Oct2014:14:54:53
12	Financials - YTD	/Enterprise Business Tree/Sales/Content/Management/	Report.BI	22Oct2014:14:56:02
13	Targets 2015	/Enterprise Business Tree/Sales/Content/Sales Targets and Actuals/	Report.BI	23Oct2014:14:56:46
14	Actuals - YTD	/Enterprise Business Tree/Sales/Content/Sales Targets and Actuals/	Report.BI	24Oct2014:14:57:01
15	Opportunities by Region	/Enterprise Business Tree/Sales/Content/Sales Targets and Actuals/	Report.BI	24Oct2014:14:57:29
16	Live Inventory	/Enterprise Business Tree/Sales/Content/Inventory/	Report.BI	24Oct2014:14:50:40
17	Inventory Projections	/Enterprise Business Tree/Sales/Content/Inventory/	Report.BI	24Oct2014:14:51:16

Output 1. Selected Columns of MDSECDS_OBJS Table

Task 3: Clean and Manage Metadata Object Descriptive Data

Depending on the naming conventions your organization follows for your metadata objects, managing the descriptive data about your objects requires a varying amount of transformation. As part of the version control process, the object name (OBJNAME) and location (LOCATION) will be used to write SAS packages to the file system. If the values for OBJNAME and LOCATION contain characters that are problematic for use in file or path names on your operating system, do some data transformations to make the values compatible with your operating system yet still recognizable as the name of the object and location.

For example, consider object and location names in your folder structure that contain spaces. Using spaces in the names of metadata objects and folders is valid and does not present any problem for use in a metadata repository or the SAS applications that use those metadata objects. However, if you write file paths and names with spaces to a unix-based operating system, referencing those file paths and names in shell scripts can be problematic. While some escaping of special characters can be accomplished, this situation can present hurdles.

For our implementation, primarily we were concerned with spaces and some special characters, such as parenthesis and slashes. Tailoring these transformations to your implementation can be done with fairly straightforward DATA step programming as shown in Code Sample 3:

```
/* Transform metadata location values */
data OBJECT_LOCATIONS_CLEANED;
   set permloc.MDSECDS_OBJS;
   length PATH_NAME_COMPRESSED $256;
   /* Remove spaces,                              */
   /* translate slashes to underscores,           */
   /* translate parenthesis to dashes.           */
   PATH_NAME_COMPRESSED=LOCATION;
   PATH_NAME_COMPRESSED=compress(PATH_NAME_COMPRESSED,' ');
   PATH_NAME_COMPRESSED=translate(PATH_NAME_COMPRESSED,'_--','/()');
run;
```

Code Sample 3. Create Object Location Values Compatible with Your File System

STEP 2: SELECT METADATA OBJECTS THAT HAVE BEEN MODIFIED

Figure 4 shows the tasks associated with selecting objects subject to the version control process.

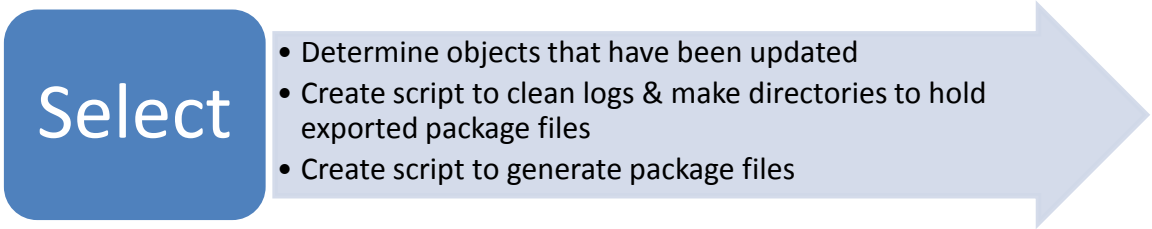


Figure 4. Metadata Object Selection Tasks

Metadata Object Descriptive Data

We've referenced the object name (OBJNAME) and object location (LOCATION) columns in the discussion about transforming metadata object descriptive data. In addition to those two columns extracted by the metadata security report macros, two additional columns in the MEDSECDS_OBJS table are important when selecting the objects to be exported for version control.

- PUBLICTYPE – shows the type of the metadata object (report, exploration, job, etc.).
- METADATAUPDATED – shows a datetime stamp for the last time the object was updated.

Task 1: Identify the Metadata Objects for Processing

Decide which types of objects you will maintain under version control and how frequently you will add versions to your version control repository. Depending on the environment (development, testing, or production) and the type of objects you use most, this may vary. At a minimum, we automate our version control for jobs, tables, and stored processes in some environments, but include report and visual exploration objects in others. We also select any object that has been updated since the previous execution of the version control process. Typically, this is every 24 hours, which gives us daily version control. But in the event of conflicting scheduled maintenance or a need to execute an "on-demand" process, we have the flexibility to catch-up or carry out additional version control processing. You could run less frequently and select objects that have been updated longer ago. This might be appropriate for application development environments that have prescribed promotion schedules that are less frequent.

In order to select metadata objects for processing based on the previous execution, you need to maintain an historical datetime indicator for the previous execution. There are many ways to write, maintain, and check a datetime stamp. We choose to simply keep a timestamp table that records the date and time when the archive process starts and the date and time when the archive process successfully completes. Maintaining a beginning and ending timestamp offers an easy recovery method in the event that the version control process is interrupted or fails. The next execution is built to process all objects updated since the **last** successful completion. So if the previous run fails, the objects selected for the previous run will be selected again, along with any newly eligible objects. Output 2 shows an excerpt of the data in our timestamp table:

Sample Excerpt from TIMESTAMP Table

Obs	SUCCESS_TIME	START_TIME
272	27OCT2014:18:33:30	27OCT2014:18:30:09
273	28OCT2014:18:34:26	28OCT2014:18:30:09
274	29OCT2014:18:33:12	29OCT2014:18:30:05
275	30OCT2014:18:33:24	30OCT2014:18:30:09

Output 2. Sample Data from Timestamp Table

The selection process then becomes a simple comparison between the values of the METADATAUPDATED column and the maximum value for START_TIME when SUCCESS_TIME is not missing. Code using the SQL procedure to apply this comparison is shown in Code Sample 4:

```
/* Select updated metadata objects */
proc sql;
   create table TO_BE_PROCESSED as
   select obj.*
   from work.OBJECT_LOCATIONS_CLEANED obj
   where PUBLICTYPE in ('Job', 'Table', 'Report.BI',
                        'Report.StoredProcess', 'VisualExploration')
     and input(obj.METADATAUPDATED,datetime20.) +
         tzoneoff('America/New_York') >
   (select max(START_TIME) from permloc.TIMESTAMP
      (where=(SUCCESS_TIME is not missing)));
quit;
```

Code Sample 4. Select Objects Eligible for Version Control

Replace or augment the values listed for PUBLICTYPE to reflect the types of metadata objects you will archive. Datetime information is stored on the metadata server as GMT values (SAS® 9.4 Metadata Model: Reference), while the timestamps displayed by your clients (Visual Analytics or Data Integration

Studio) are usually set to show local time. To be consistent with the times seen by people using the metadata objects and the archived entries, use the TZONEOFF function to adjust the cutoff time for objects to use the same time zone as displayed in the data.

Shell Scripting

The actions of the version control process are executed via a series of shell scripts. These scripts can be written for the operating system of your choice using appropriate scripting languages for that operating system, such as Korn shell for UNIX or Windows PowerShell for Windows. The code samples in this paper use Korn shell and have been run under SunOS and Linux.

SAS jobs generate the code for the scripts. Using a NULL DATA step (a DATA step with the keyword _NULL_ as described in SAS(R) 9.4 Language Reference: Concepts, Fourth Edition), the SAS job writes the code for the scripts to the file system. By iterating through the TO_BE_PROCESSED table, lines of code to package and archive the selected objects are generated. A location for the resulting scripts (in this case a /bin or /scripts) directory must exist with appropriate permissions so that the username under which the SAS jobs are executed can write to it.

Task 2: Generate a Script to Clean out the Log Files from Previous Executions

Use the log files created during the version control process to determine success or failure for the current process during later error checking and reporting tasks. Retain the logs for a short period of time so they can be examined should problems occur. Beyond that, clean out logs older than the duration of your planned log retention period.

For our version control process, we retain log files for 7 days. Create the script to manage log files using a NULL DATA step as shown in Code Sample 5:

```
/* Create text date name for use in log directory name           */
data _null_;
   NOW='20'||put(today(),yymmdd6.);
   call symput('dirdate',NOW);
run;
/* FILE statement specifies location of the resulting script.   */
/* PUT statements write to the script location.                 */
/* # begins comment lines                                       */
/* echo writes a message to the script log                      */
/* cd changes directory                                         */
/* find and rm locates and removes any logs and directories     */
/*    older than 7 days                                         */
/* conditionally (if exists) removes logs with today's date     */
/* creates a log directory for today                            */
data _null_;
   file '/local/bin/Clean_Logs' filename=OUT;
   put '#!/bin/sh';
   put '# Clean out the logs for the version control process.';
   put '# Remove log files and directories older than 7 days.';
   put '# Create directory for the date of the current process.';
   put '#';
   put 'echo Begin Clean_Logs script.';
   put "cd /local/VC/logs/";
   put "find /local/VC/logs/ -mtime +6 -exec rm -r {} \;";
   put "if [ -d &dirdate    ]";
   put 'then';
   put "rm -r &dirdate";
   put 'fi';
   put "if [ ! -d &dirdate  ]";
   put 'then';
   put "mkdir &dirdate";
```

```
    put 'fi';
run;
```

Code Sample 5. Script to Manage Logs

This sample uses the file locations and names as examples. Change these (/local/bin/Clean_Logs, /local/VC/logs) to appropriate values for your environment. The code includes writing comments to the shell script to explain the purpose of the statements.

An option (not presented in the code sample, but one that we use) is to insert a subdirectory name in the log's file path based on the hostname or some other identifier for the environment in which the version control process is running. Code Sample 6 shows the substitution of a hostname in the directory file path:

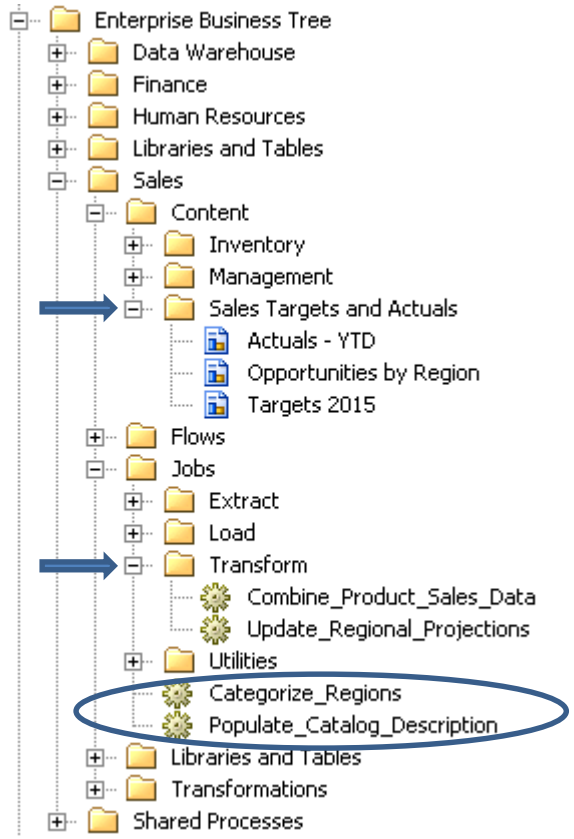
```
put "cd /local/VC/&hostname/logs";
```

Code Sample 6. Use a Hostname in Your File Path

Then, if you run the version control process in multiple environments, you can set the value of &HOSTNAME to represent the environment identification. This enables you to keep the logs from each environment centrally located but separate from one another. It is a useful option when writing log files to a location that is not local to your server host.

Task 3: Generate a Script to Create the Directories Needed to Hold the Exported Package Files

For ease of management, we represent each metadata object folder location as a separate directory on the file system and the version control repository, but they are not organized hierarchically. Each of these directories will hold all of the metadata objects selected for version control processing from the corresponding metadata location. Use Figure 5 to see the mapping between the original metadata object location and the file system (and version control repository) location:



Metadata object location (folder)	→	**File system location (directory)**
/Enterprise Business Tree/Sales/Content/Inventory	→	EnterpriseBusinessTree_Sales_Content_Inventory

/Enterprise Business Tree/Sales/Content/Management	→	EnterpriseBusinessTree_Sales_Content_Management
/Enterprise Business Tree/Sales/Content/Sales Targets and Actuals	→	EnterpriseBusinessTree_Sales_Content_SalesTargetsandActuals
/Enterprise Business Tree/Sales/Jobs/Extract	→	EnterpriseBusinessTree_Jobs_Extract
/Enterprise Business Tree/Sales/Jobs/Load	→	EnterpriseBusinessTree_Jobs_Load
/Enterprise Business Tree/Sales/Jobs/Transform	→	EnterpriseBusinessTree_Jobs_Transform
/Enterprise Business Tree/Sales/Jobs/Utilities	→	EnterpriseBusinessTree_Jobs_Utilities
/Enterprise Business Tree/Sales/Jobs	→	EnterpriseBusinessTree_Jobs

Figure 5. Metadata Object Locations Mapped to File System Locations

The individual package files for the three metadata objects in the */Sales/Content/Sales Targets and Actuals* folder will be placed in the *EnterpriseBusinessTree_Sales_Content_SalesTargetsandActuals* directory on the file system. Similarly, the two metadata objects in the */Sales/Jobs/Transform* folder will be placed in the *EnterpriseBusinessTree_Sales_Jobs_Transform* directory on the file system. The same pattern of correspondence applies for the other subfolders in the */Sales/Content* and */Sales/Jobs* folders. However, package files for the two metadata objects (circled) that are directly in the */Sales/Job* folder will be placed in *EnterpriseBusinessTree_Sales_Jobs*. Although the */EnterpriseBusinessTree_Sales_Jobs* directory is a separate directory, the subdirectories with Extract, Load, Transform, and Utilities in the names are not subordinate on the file system, even though they are subfolders in the metadata object folder tree.

To prepare for exporting the metadata objects as SAS package files, the next script manages the file system directories that will hold the package files. This script, using the metadata descriptive data in the TO_BE_PROCESSED table, identifies the unique folder names containing the objects, checks to see if a directory corresponding to that folder exists on the file system, and if not, creates it. Create the script to manage the file system directories using a NULL DATA step as demonstrated in Code Sample 7:

```
/* Collect unique directory names for the file system.                */
proc sort data=TO_BE_PROCESSED(keep=PATH_NAME_COMPRESSED
   where=(upcase(PATH_NAME_COMPRESSED) ^contains 'DEPLOYED'))
   out=UNIQUE_FOLDER_NAMES(keep=PATH_NAME_COMPRESSED) nodupkeys;
   by PATH_NAME_COMPRESSED;
run;
data _null_;
   set UNIQUE_FOLDER_NAMES;
   file '/local/bin/Make_Directories' filename=OUT;
   m=-1;
   if _n_=1 then do;
      put '#!/bin/sh';
      put '# Create folders for the version control process.';
      put "echo Begin Make Directories script";
      put "cd /local/VC/folders/";
   end;
   put '#';
   put 'if [ ! -d '  PATH_NAME_COMPRESSED ' ]';
   put 'then';
   put ' mkdir ' PATH_NAME_COMPRESSED;
   put 'fi';
   *put "rm /local/VC/folders/" PATH_NAME_COMPRESSED +m '/*.spk';
   put '#';  */
run;
```

Code Sample 7. Create Script for Managing File System Directories

Note the WHERE clause on the PROC SORT statement. We follow a standard that places all deployed job objects in folders with the name “DEPLOYED” and we do not want to include deployed job objects in our archive. So by filtering with the WHERE clause, we easily exclude those objects. While this specific filter might not apply to your implementation, the example is included to show how easily you can control your version control process with straightforward SAS code manipulating the columns of descriptive data collected by the %MDSECDS macros.

Another optional task shown in this code sample is removing all of the existing package files from the file system directory prior to writing new package files. That is shown with the commented-out PUT statement. We maintain the last package file on the file system so that we have the option of retrieving the last version from the file system or the version control system repository. Versions prior to the last can only be found in the version control system repository. If you include this optional code, take care to remove only the exported package files and not the entire directory structure. Often version control software requires the presence of configuration files that contain the history of activity related to the file, so code to remove exported package files from the file system will vary based on the version control software used.

Task 4: Create Script to Generate Exported Package Files

The purpose of the next script to be generated (with a NULL DATA step) is to create the exported package files for each object. The package files are created using the Batch Export Tool (documented in SAS® 9.4 Intelligence Platform: System Administration) that is supplied as part of the SAS® 9.4 Intelligence Platform. That means that the code we generate is a call to the Batch Export Tool for each of the metadata objects selected for version control processing.

Batch Export Tool

The syntax for the Batch Export Tool contains many options, but it can be used with only a few options specified. Table 1 enumerates each option used and its purpose for a single invocation of the Batch Export Tool for one metadata object:

Batch Export Tool Syntax and Options	Purpose
ExportPackage	Invoke the Batch Export Tool
-profile “location of connection profile file”	Provide a location for finding the connection profile to connect to the metadata repository.
-package “location of resulting package file”	Provide a location for writing the package file to the file system.
-objects “location of the metadata object”	Provide the location and metadata object name to be exported.
-log “location for export process log file”	Provide a location for writing a log file for the export process to the file system.

Table 1. Batch Export Tool Syntax and Options

Many topics are covered in the documentation for the Batch Export Tool. This paper highlights two key issues to help you quickly accomplish your implementation.

1. Connection Profile – provides the credentials for connecting to the Metadata Server to run the Batch Export Tool.
2. Windowless (headless) X Server – provides a mechanism for running the Batch Export Tool without a display.

Connection Profile

A connection to a metadata server requires a connection profile that specifies such information as the machine housing the metadata sever, the port for connection, and credentials needed for authorization to access the metadata. While the documentation shows that the information in a connection profile can be specified as parameters when invoking the Batch Export Tool, it is more straightforward to create and store a connection profile file that contains the necessary information and simply reference the profile when invoking the Batch Export Tool.

The process for creating a connection profile is covered in the SAS® 9.4 Intelligence Platform: Desktop Application Administration Guide. The easiest way to create a connection profile file is to use the connection profile wizard from one of the client applications such as SAS Management Console or SAS Data Integration Studio. By default, connection profiles are saved to

/Users/username/AppData/Roaming/SAS/MetadataServerProfiles. Output 3 shows example contents of a connection profile file created via a client application:

```
#Properties file updated on: Wed Nov 05 10:54:05 EST 2014  !!!!! DO NOT
EDIT !!!!!!!
#Wed Nov 05 10:54:05 EST 2014
SingleSignOn=false
AllowLocalPasswords=true
port=8561
SelectedReps="repository.objectid"
InternalAccount=false
Name="profile_name"
userid="username_to_connect_with"
authenticationdomain=
password="encrypted_password"
host="fully.qualified.host.name"
```

Output 3. Connection Profile Contents

While it might seem fairly straightforward to simply copy and edit, the recommended method is to use the connection profile wizard in one of the Intelligence Platform clients. Then copy that connection profile to the location required by your version control process.

Windowless (headless) X Server

Due to internal coding requirements of the Batch Export Tool, the specification of a DISPLAY environment variable is required. In order to use a DISPLAY environment variable, an X Server must be present. Since the script is running in a batch or non-windowed environment, it is necessary to execute the script within a windowless X Server. We have been successful using the Xvfb virtual framebuffer X server for this purpose. If you are unfamiliar with Xvfb, you might need to consult the system administrator for the host on which your version control process will run in order to have the Xvfb server installed or configured. Code to invoke Xvfb, as well as the Batch Export Tool for each metadata object to be packaged, is shown in Code Sample 8:

```
   /* Export each metadata object to a package file.         */
   data _null_;
      set TO_BE_PROCESSED end=DONE;
      file "/local/bin/Create_Package_Files" filename=OUT;
      m=-1;
      if _n_=1 then do;
         put '#!/bin/sh';
         put "echo Begin Create Package Files script.";
         /* Start X server */
         put '/usr/openwin/bin/Xvfb :1 2>/dev/null &';
         put 'sleep 20';
         put 'DISPLAY=localhost:1; export DISPLAY';
      end;
      put '#';
      put '/opt/sas94/SASHome/SASPlatformObjectFramework/9.4/ExportPackage
\';
      put "-profile /local/VC/ \" ;
      put "-package /local/VC/folders/"
          PATH_NAME_COMPRESSED +m '/' OBJECT_NAME_COMPRESSED +m '.spk \';
      put '-objects "' LOCATION +m OBJNAME +m '(' PUBLICTYPE +m ')" \';
      put "-log /local/VC/logs/&dirdate/" PATH_NAME_COMPRESSED +m '-'
          OBJECT_NAME_COMPRESSED +m '.log';
      if DONE then do;
```

```
      put '#';
            put 'sleep 20';
            put '/bin/pkill Xvfb';
      end;
   run;
```

Code Sample 8. Create Script to Export Packages

STEP 3: PACKAGE MODIFIED METADATA OBJECTS

Figure 6 lists the tasks associated with exporting metadata objects as SAS package files.

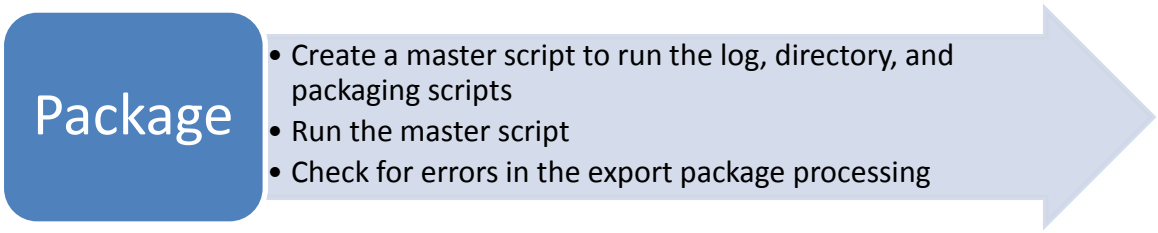


Figure 6. Package Modified Metadata Objects Tasks

Task 1: Create a master script to run the log, directory, and packaging scripts.

Next create a master script that calls the previously created scripts (that clean logs, make directories, and package metadata objects). At this point, in addition to creating the script (with a NULL DATA step), the SAS job will execute the script. Code Sample 9 shows generating the master script for the first half of the version control process:

```
   /* Create and run the master script */
   data _null_;
      file '/local/bin/Version_Control_Script1' filename=OUT;
      put '#!/bin/sh';
      put 'echo Begin Version Control Script 1.';
      put "date '" '+DATE: %m/%d/%y TIME:%H:%M:%S' "'";
      put '# Clean the logs';
      put '# Make the directories';
      put '# Create the packages';
      put '#';
      put 'cd /local/bin';
      put '/local/bin/Clean_Logs >  /local/VC/logs/Clean_Logs 2>&1';
      put '/local/bin/Make_Directories >  /local/VC/logs/Make_Directories
          2>&1';
      put '/local/bin/Create_Package_Files >
          /local/VC/logs/Create_Package_Files.log 2>&1';
      put '#';
   run;

   %sysexec /local/bin/Version_Control_Script1 >
/local/VC/logs/Version_Control_Script1.log
            2>&1;
```

Code Sample 9. Create and Run Master Script 1

Task 2: Run the master script

Use the %SYSEXEC macro from within SAS to execute the master script. There are a variety of ways to issue operating system commands from within a SAS job. In our environment, using %SYSEXEC to execute the script from a scheduled SAS batch job worked best (as shown above).

On many operating systems, files must have specific permissions or characteristics in order to be executable (run as a command). You can address this issue by setting the permissions within your SAS

job (or session). However, an easy way to handle this issue is to set the permissions or characteristics manually after the file is written the first time. Subsequently, the file can continue to be overwritten and executed as needed. It is not necessary to generate the master script during each version control process since the commands to run the subordinate scripts do not change. For consistency related to development and maintenance, we chose to regenerate the master script as part of the daily processing.

Task 3: Check for Errors in the Export Packaging Process

The Batch Export Tool produces logs as it packages each metadata object. Use a SAS job to parse those logs and identify warnings and error messages. If warning or error messages are found, generate and send an e-mail alert including specific information about the errors found. Checking the value of SYSRC for the %SYSEXEC call might appear to solve the problem of checking for warnings or errors for the execution of the Batch Export Tool. In fact, it does not. The value of SYSRC is simply the return code for the execution of the master script, not the scripts within the master script. Code Sample 10 and Code Sample 11 show an example to determine whether log files are present and methods for parsing the logs for warning and error messages:

```
/* Point to the log location for this version control process. */
filename DIRLIST pipe "ls /local/VC/logs/&dirdate/*.log";
data DIRLIST;
   infile DIRLIST lrecl=200 truncover;
   input LINE $200.;
   /* If there is a 'No' message returned from checking        */
   /* the log files, then there are no logs to process.       */
   if index(LINE,'No such file or directory')
      then call symput('LOGCOUNT','0');
   else call symput('LOGCOUNT','1');
run;
```

Code Sample 10. Check for existence of log files

```
/* Read the logs from the Batch Export Tool and look for warnings      */
/* and errors. Write a report of the findings and email an alert      */
/* message, if found.                                                 */
%global LOGCOUNT WARNS_ERRS_COUNT;
%macro process_logs;
   %if &LOGCOUNT ne 0 %then %do;
      filename IN "/local/VC/logs/&dirdate/*.log";
      /* Read the logs and set a warning/error status for each line.  */
      data MESSAGES;
         length LF $500 LOG_FILENAME $500 STATUS 8;
         infile IN truncover filename=LOG_FILENAME end=DONE;
         input TEXT $500.;
         LF=LOG_FILENAME;
         if substr(TEXT,1,4) in ('WARN', 'ERRO') then STATUS=1;
         else STATUS=0;
      run;
      data WARNS_AND_ERRS;
         set MESSAGES(where=(STATUS ne 0));
      run;
      ODS listing close;
      ODS html body="Export_Package_Warnings_and_Errors_Report.html"
          style=seaside path="/local/VC/Error_Reports";
      /* Header for report                                             */
      data _null_;
         file print;
         put "<div class='c SystemTitle'>Export Package Warnings and
             Errors Report </div>";
         put "<div class=
```

```
            'c SystemTitle'>%sysfunc(datetime(),nldatmap.)</div>";
      run;
      proc sort data=WARNS_AND_ERRS;
         by LF TEXT;
      run;
      proc print data=WARNS_AND_ERRS label double;
         id LF;
         var TEXT ;
         label LF='Export Package Log File Location'
               TEXT='Warning or Error Message'
      run;
      proc sql noprint;
         select count(*) into: WARNS_ERRS_COUNT from WARNS_AND_ERRS;
      quit;
   %end;
%mend;
%process_logs;
/* If WARNS_ERRS_COUNT is not 0, then send e-mail alert.              */
```

Code Sample 11. Check Logs for Errors and Warnings and Conditionally Send E-mail Alert

STEP 4: UPDATE VERSION CONTROL REPOSITORY

Figure 7 shows the tasks associated with updating the version control repository with the exported SAS package files.

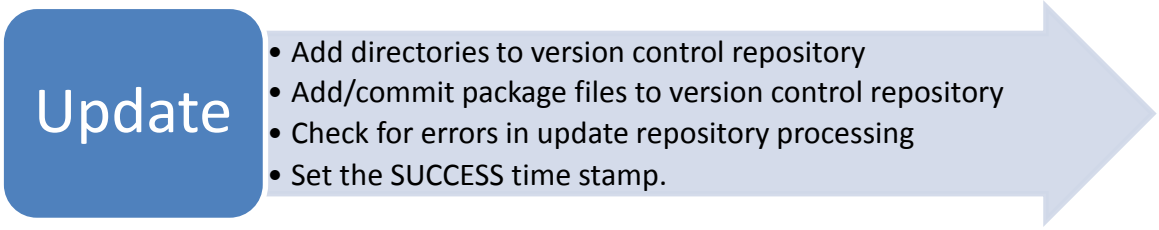


Figure 7. Update Version Control Repository

Version Control Software

The choice of version control software is yours. The only requirement is that the software can be used via batch or line command access. Since all of the actions used are done via scripting, software that can be accessed only via a GUI will not be usable. For our version control process, we used CVS. This choice was primarily based on previous use and compatibility with other processes in place, as well as ease of use.

Set up of the version control repository and access methods are not covered in this paper. However, code samples contain the code used and scripts generated so the actual line commands used to add the package files to the CVS repository are available. The previous link to CVS documentation contains several excellent sources of information (look at the External Links section). In particular, read information about initializing CVS (cvs init) and setting the CVS root.

Task 1: Add Directories to Version Control Repository

During the second stage of the version control process, updating the version control repository, you will again create scripts to perform the necessary tasks by using SAS jobs with a NULL DATA step. The first script adds directories to the CVS repository. As explained in Figure 5, each directory in the version control repository corresponds to the subfolder name in which the metadata object resides. Code Sample 12 shows the SAS code to generate a script to add directories to a version control repository:

```
/* Create script to make sure there is a corresponding directory  */
/* in the version control repository for each unique location     */
/* (folder structure) in the metadata object tree.               */
```

```
   data _null_;
      set UNIQUE_FOLDER_NAMES;
      file '/local/bin/Add_Directories_To_CVS' filename=OUT;
      m=-1;
      /* Write lines to begin the script.                              */
      if _n_=1 then do;
         put '#!/bin/sh';
         put '# Add folder to the VC repository for each metadata folder.';
         put 'echo Begin Add Directories To CVS script.';
         put 'export
CVSROOT=:pserver:username@host.name.com:/cvs/repository_location';
         put 'cd /local/VC/';
         /* The batch process to add folders is more successful if a       */
         /* checkout is done before beginning the add commands.           */
         put "/usr/local/bin/cvs checkout folders/CVS_placeholder.txt";
         put "cd /local/VC/folders/";
      end;
      put '#';
      put '/usr/local/bin/cvs add '  PATH_NAME_COMPRESSED ;
   run;
```

Code Sample 12. Add Directories to Version Control Repository

Task 2: Add/commit Package Files to Version Control Repository

Add and commit the exported package files to the version control repository. Terminology may differ for these actions, depending on the version control software used. Use a NULL DATA step to generate the script to contain the version control commands to place the newest version of the package file in the version control repository. Code Sample 13 shows an example of generating the script for each package file:

```
   /* Create script to add and commit each exported package file */
   /* to the version control repository.                        */
   data _null_;
      set TO_BE_PROCESSED end=done;
      file "/local/bin/Add_Objects_To_CVS" filename=OUT;
      m=-1;
      /* The batch process to add files is more successful if a     */
      /* checkout is done before beginning the add commands.       */
         if _n_=1 then do;
            put '#!/bin/sh';
            put "Begin Add Objects To CVS script.";
            put 'export
CVSROOT=:pserver:username@host.name.com:/cvs/repository_location';
            put 'cd /local/VC/';
            put "/usr/local/bin/cvs checkout folders/CVS_placeholder.txt";
         end;
         put '#';
         put "cd /local/VC/folders/";
         put 'cd ' PATH_NAME_COMPRESSED ;
         put '/usr/local/bin/cvs add ' OBJECT_NAME_COMPRESSED +m '.spk ';
         put "/usr/local/bin/cvs commit -fm '" OBJNAME +m "' "
OBJECT_NAME_COMPRESSED +m '.spk ';
   run;
```

Code Sample 13. Add Package Files to Version Control Repository

Task 3: Set up Error Checking for Version Control Processing

The scripts that add directories and package files to the version control repository create logs of their activity. Like the error check job following the export package process, use a SAS job to parse the resulting logs and if errors are found, send an e-mail alert. In addition, a report that lists the results of the entire process (objects packaged and archived) is maintained for quick and easy reference. This report is a file to which the day's version control activities are appended by extracting lines from the original processing logs. Code Sample 14 shows the simple steps to generate the report file:

```
/* Generate the Check Results script.                        */
data _null_;
   file "/local/bin/Check_Results" filename=out;
   put '#!/bin/ksh';
   put '# This script checks the results for version control process.';
   put 'cd /local/bin';
   put 'print "Number of objects to be packaged and archived"';
   put "grep 'objects' Create_Package_Files | wc -l";
   put "cd /local/VC/logs";
   put 'print "Number of objects packaged"';
   put "grep 'finished successfully' Create_Package_Files.log | wc -l";
   put 'print "Number of objects archived"';
   put "grep 'Checking in' Add_Objects_To_CVS | wc -l";
   put 'cd /local/bin';
   put 'print "Objects to be packaged and archived"';
   put "grep 'objects' Create_Package_Files ";
   put "cd /local/VC/logs";
   put 'print "Objects packaged"';
   put "grep 'Exporting objects to package:' Create_Package_Files.log";
   put 'print "Objects archived"';
   put "grep 'Checking in' Add_Objects_To_CVS";
run;
```

Code Sample 14. Create Script to Show Overall Results for Version Control Process

Task 4: Run master script 2.

Create and run a master script to execute the previously generated scripts for updating the version control repository and check for errors. Again, it is not necessary to regenerate this master script with each process, but we leave it as part of the daily process just for consistency's sake. Code Sample 15 shows the code to generate and execute the second master script:

```
   /* Write script to execute the version control command scripts. */
   data _null_;
         file "/local/bin/Version_Control_Script2" filename=OUT;
         put '#!/bin/sh';
         put 'echo Begin Version Control Script2.';
         put '# Add the directories to version control repository';
         put '# Add the objects to version control repository';
         put '# Check for errors and report on VC process';
         put '#';
         put 'cd /local/bin';
         put ". ./Add_Directories_To_CVS >
/local/VC/logs/Add_Directories_To_CVS 2>&1";
         put 'cd /local/bin';
         put ". ./Add_Objects_To_CVS >  /local/VC/logs/Add_Objects_To_CVS
2>&1";
         put 'cd /local/bin';
         put ". ./Check_Results >> /local/VC/logs/Check_Results 2>&1";
   run;
```

```
%sysexec %str(cd /local/bin; . ./Version_Control_Script2 >
         /local/VC/logs/Version_Control_Script2 2>&1);
```

Code Sample 15. Generate and Execute Second Master Script

Task 5: Set the Ending Timestamp

Lastly, set the ending (SUCCESS_TIME) timestamp for the completion of the version control process. Run a SAS job to generate a corresponding ending time to the beginning timestamp as previously shown in Code Sample 4. Since the original code in Code Sample 4 selects a record where SUCCESS_TIME is not missing, by adding the ending time, the next execution of the version control process will begin with a new selection process. Code Sample 16 shows the data step to add the ending timestamp:

```
/* Add the success timestamp to control object selection for archiving. */
data permloc.TIMESTAMP;
   set permloc.TIMESTAMP;
   if SUCCESS_TIME=. Then SUCCESS_TIME=datetime();
run;
```

Code Sample 16. Set the Ending Timestamp

CONCLUSION

In the version control process we've implemented, every metadata object (of the types of interest) is first packaged and committed to our version control repository to initialize the system. This initialization runs in the same way as a daily update; it simply selects all objects. Then, as objects are updated, the newest version continues to be added to the repository. The latest version is also maintained on the file system so that developers can instantly retrieve the most recent version. A brief document explaining what is included in the version control process, when it runs, and how to retrieve versions is available to guide content authors and applications developers to quickly restore a damaged Visual Analytics report or DI job. Besides the time saved by not having to recreate work, the smiles on their faces and relief in their voices when a working version is recovered demonstrates the value of an automated version control process.

REFERENCES

SAS® 9.4 Companion for UNIX Environments, Fourth Edition. Cary, NC: SAS Institute Inc.

SAS® 9.4 Intelligence Platform: Desktop Application Administration Guide, Fourth Edition. Cary, NC: SAS Institute Inc.

SAS® 9.4 Intelligence Platform: System Administration Guide, Third Edition. Cary, NC: SAS Institute Inc.

SAS® 9.4 Intelligence Platform: Security Administration Guide, Second Edition. Cary, NC: SAS Institute Inc.

SAS® 9.4 Language Reference: Concepts, Fourth Edition. Cary, NC: SAS Institute Inc.

SAS® 9.4 Metadata Model: Reference. Cary, NC: SAS Institute Inc.

Wikipedia, Concurrent Versions System. Accessed January 16, 2015. http://en.wikipedia.org/wiki/Concurrent_Versions_System.

Wiggins, David P. XVFB. Accessed January 16, 2015. http://www.x.org/archive/current/doc/man/man1/Xvfb.1.xhtml.

CONTACT INFORMATION

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

Jerry Hosking

Enterprise Solutions Data & Analytics, SAS Institute
919-531-7056
jerry.hosking@sas.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.