SAS® GLOBAL FORUM 2018

USERS PROGRAM

Using UNIX Shell Scripting to Enhance Your SAS Programming Experience

By James Curley

April 8 – 11 | Denver, CO
#SASGF

ELIASSEN GROUP®
Biometrics & Data Solutions

Using UNIX Shell Scripting to Enhance Your SAS Programming Experience

By James Curley

ABSTRACT

This series will address three different approaches to using a combination of UNIX shell-scripting and SAS® programming to dramatically increase programmer productivity by automating repetitive, time-consuming tasks.

Program One embeds an entire SAS® program inside a UNIX shell script, feeds it an input file of source and target data locations, and then writes the SAS® copy program out to each directory with dynamically updated LIBNAME statements. This approach turned a 25 hour job into mere minutes.

Program Two of the series reviews another deploy shell script that creates its own input file and determines whether a standalone SAS® program should to be deployed to each directory. This approach turned an 8 hour job into just a couple minutes.

Program Three consists of a smaller shell script that dynamically creates SAS® code depending on the contents of the directory in which the program is executed.

At the end of these three segments, you will have a better understanding of how to dramatically increase the productivity of your SAS® programs by integrating UNIX shell-scripting into your SAS® programming to automate repetitive, time-consuming tasks. None of these programs requires any input from the user once started, so the only programming required is the few minutes it takes to set up the programs.

PROGRAM 1 - deployDataCopy.ksh

The first program uses a UNIX shell script to remove the human element from the task of copying a simple SAS program into 500 directories, updating the LIBNAME statements, executing the program and checking the logs for errors. A UNIX shell script is the perfect delivery system to do all of that work in a fraction of the time.

Here is a sample input file, trimmed down to 3 lines for the purposes of this illustration. (We typically have larger files to facilitate the loading of data for hundreds of studies at once). Each line consists of two parts: the protocol name and the path to the source datasets, delimited by a caret:

```
X3351001^/Volumes/app/data/prod/X335/nda1/X3351001/data
X3351002^/Volumes/app/data/prod/X335/nda_final/X3351002/data
X3351007^/Volumes/app/data/prod/X335/csr1/X3351007/data
```

To write out the SAS program the UNIX cat command is used to create a file in the program directory called DataCopy.sas. The cat command will write everything in this program between the two instances of the word “FINAL”.

```
cat > DataCopy.sas << FINAL
/* Put the entire SAS program here */
FINAL
```

cat – Not just a useful UNIX command

Using UNIX Shell Scripting to Enhance Your SAS Programming Experience

By James Curley

ELIASSEN GROUP® Biometrics & Data Solutions

PROGRAM 1 - CONTINUED

The key advantage to embedding the SAS program inside the UNIX shell script is that the UNIX variables can be referenced in the SAS® program and will resolve to their UNIX variable values when the SAS program is written out to a file:

```
/** Set the location for source & target  **/
%let inpath =${s_path};
%let outpath=${d_path};
/** Set the dataset flags **/
%let demog  = ${de_flag};
%let drug   = ${dg_flag};
%let random = ${ra_flag};
```

As a result, the variable names as they appear in the shell script (above) will be the resolved variable values in the written SAS program (below):

```
/** Set the location for source & target  **/
%let inpath =/Volumes/app/data/prod/X335/nda1/X3351001/data;
%let outpath=/Volumes/app/data/prod/prjX335/pbrer2017/X3351001/data;
/** Set the dataset flags **/
%let demog  = YES;
%let drug   = YES;
%let random = YES;
```

PROGRAM 1 - CONCLUSION

So there you have it: A SAS program embedded in a UNIX shell script, deployed across any number of directories, executed, and then checked for errors. The hours this one utility has saved my team is almost inconceivable. What makes this combination more functional than running either a shell script or SAS program separately is that embedding the SAS program inside the UNIX shell script allows you the use of UNIX variables in the SAS program that will resolve to their UNIX variable values when the SAS program is stored prior to execution. Being able to dynamically update the SAS program with each iteration of the input file loop thus makes this combination especially useful.

The time savings is very easy to calculate for this illustration. Performing the tasks manually takes 25 to 40 hours of work. To set up and run this shell script takes 15 minutes. The program may take an hour to run, but the programmer need only start the program and move on to other tasks.

In program 2 of this series I will show how to deploy a standalone SAS program across the same 500 protocols; however, in this example, the program will create its own input file and then determine on a protocol-by-protocol basis if it needs to be deployed or not.

Using UNIX Shell Scripting to Enhance Your SAS Programming Experience

By James Curley

PROGRAM 2 - deployTreatments.ksh

Program two also uses a UNIX shell script to remove the human element from the task of copying a simple SAS program into 500 directories, executing the program and checking the logs for errors. Since the program resides in the same directory as the dataset it is analyzing, there are no LIBNAME statements to update. This means the program doesn't have to be embedded in the UNIX shell script, and can instead be deployed as a standalone program using only the shell script to quickly deliver it where it needs to be.

This section of code creates an input file of all the treatment datasets in the submission. This file is then piped into a grep that removes all the symlinks before writing it out to the deploy1.tmp temporary file. The sed statement will turn any number of spaces into a single space and allow us to use a space delimiter when the file is read a little further down in the code:

```
ls -l /Volumes/app/data/prod/prj${proj}/${sub}/*/data/treatment.sas7bdat |
grep ^- > ./deploy1.tmp


sed 's/[ ]\{1,\}/ /g' ./deploy1.tmp > ./deploy.lst
```

The program will now use the cat command and while loop to process the file one line at a time until reaching the end. The first variable assignment is going to remove the entire path and file name from the ls-l and store that value in ${line2}. Next, the protocol name will be cut from the ${line2} and assigned to ${prot}. Lastly, the dirname statement will remove the filename from ${line2}, leaving just the data directory path assigned to ${path}.

PROGRAM 2 – CONTINUED

```
cat ./deploy.lst |
  while read line
  do

  line2=`echo ${line} | cut -d' ' -f9`
  prot=`echo ${line2} | cut -d'/' -f8`
  path=$(dirname ${line2})
```

Next the program will determine if the Listing from the previous deployment is still "current". It would be wasteful to deploy a fresh program in each study where the underlying data hasn't even changed. This is accomplished by doing an ls -1 –t into a temporary file, which will produce a reverse chronological listing of the directory contents. Reading this list will allow us to set a flag telling us whether the listing is out-of-date and only regenerate the listing when the data has actually changed:

Using UNIX Shell Scripting to Enhance Your SAS Programming Experience

By James Curley

ELIASSEN GROUP®
Biometrics & Data Solutions

PROGRAM 2 - CONTINUED

```
 ls -1 -t > ./dir_lst.txt

  flag=no
  cat ./dir_lst.txt |
    while read line1
    do

    if [[ ${flag} = "yes" ]]; then
      continue
    elif [[ ${line1} = "treatment.sas7bdat" ]]; then
      echo -e "\n${file_name} is either out of date or does not exist....deploying"

      #  Check for and remove any existing log, listing and SAS® files.
      [[ -e ./${file_name}.lst ]] && rm ./${file_name}.lst
      [[ -e ./${file_name}.log ]] && rm ./${file_name}.log
      [[ -e ./${file_name}.sas ]] && rm ./${file_name}.sas
      #  Copy the program to the target directory and execute SAS® program.
      cp ${src_file} ${file_name}.sas

      echo -e "Running ${file_name}.sas for protocol ${prot}"
      sas94 -log ./${file_name}.log -print ./${file_name}.lst -nodms
./${file_name}.sas
      flag=yes
    elif [[ ${line1} = "${file_name}.lst" ]]; then
      echo -e "\n${file_name} listing is current for protocol ${prot}"
      flag=yes
    fi
    done
done
```

PROGRAM 2 – CONCLUSION

As you can see, it is just as simple to deploy a standalone SAS® program using a UNIX shell script as it is to embed one in the program. The two advantages to using the shell script are:

- Programmer setup time is reduced since the utility creates its own input file.
- Total run-time is reduced since the program only regenerates listings which are out-of-date.

The time savings is very easy to calculate for this illustration: Performing the tasks manually takes 8 hours of work versus a 15 second setup time for the shell script. The program may take an hour to run, but the programmer need only start the program and move on to other tasks.

Using UNIX Shell Scripting to Enhance Your SAS Programming Experience

By James Curley

ELIASSEN GROUP® Biometrics & Data Solutions

PROGRAM 3 - unpack_xpt.ksh

Clinical trial data doesn't always come in SAS® datasets, it can come in a wide range of forms from delimited text files, to excel spreadsheets to SAS® export files. In all of these cases, these raw data files need to be turned into SAS® datasets. This program will dynamically write out the LIBNAME statements for each export file in the directory so the SAS® program can iterate through them and unpack each export file.

This line of code creates an input file of all xpt files in the current directory:

```
 # Create the input list of all export files
ls -1 ./*.xpt > ./setup.lst
```

The program is to iterate through the setup file it just created and write out to a temp file, a LIBNAME statement for each xpt file on the list. The loop counter will iterate 1 more time than we need so the ((x=x-1)) line will bring the counter back to the correct number of LIBNAME statements:

```
cat ./setup.lst |
  while read line
  do
   echo -e "libname xpt${x} xport \"${line}\";" >> ./temp.txt
     ((x=x+1))
done
((x=x-1))
```

PROGRAM 3 – CONCLUSION

Here we see how UNIX shell scripts can write customized SAS® programs with information gathered on its own. This program takes just a few seconds to get setup and run, saving the programmer the time of manually writing out the LIBNAME statements for each file in the directory.

This three parts of this series demonstrated just a few different ways that UNIX shell scripting and SAS® can be used together to dramatically increase the productivity of SAS® programs by automating time-consuming, repetitive tasks.

CONTACT INFO

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

Name: Jim Curley

E-mail: JCurley@eliassen.com

http://www.jimcurley.net/sas-global-forum-2018

UNIX and SAS working together

SAS® GLOBAL FORUM 2018

April 8 – 11 | Denver, CO
Colorado Convention Center

#SASGF

Paper 2412-2018

Using UNIX Shell Scripting to Enhance Your SAS Programming Experience

James Curley, Eliassen Group

ABSTRACT

This series will address three different approaches to using a combination of UNIX shell-scripting and SAS programming to dramatically increase programmer productivity by automating repetitive, time-consuming tasks.

Part One embeds an entire SAS program inside a UNIX shell script, feeds it an input file of source and target data locations, and then writes the SAS copy program out to each directory with dynamically updated LIBNAME statements. This approach turned a 25 hour job into mere minutes.

Part Two of the series reviews another deploy shell script that creates its own input file and determines whether a standalone SAS program should to be deployed to each directory. This approach turned an 8 hour job into just a of couple minutes.

Part Three consists of a smaller shell script that dynamically creates SAS code depending on the contents of the directory in which the program is executed.

At the end of these three segments, you will have a better understanding of how to dramatically increase the productivity of your SAS programs by integrating UNIX shell-scripting into your SAS programming to automate repetitive, time-consuming tasks. None of these utilities requires any input from the user once started, so the only effort required is the few minutes it takes to set up the programs.

INTRODUCTION

The essence of this entire paper is the idea of using UNIX shell scripting to try to remove the human element from the time consuming, repetitive tasks of delivering SAS programs to multiple directories, updating LIBNAME statements, executing SAS programs and then checking the logs for errors. The shell scripts can even go as far as creating its own input files and deciding for the programmer if the SAS program needs to be copied into a directory or not. A UNIX shell script is the perfect delivery system to do all of that work, and more, in a fraction of the time a programmer could do it manually.

PART ONE - EMBEDDING A SAS PROGRAM IN A UNIX SHELL SCRIPT TO DEPLOY IT ACROSS MULTIPLE DIRECTORIES

This paper will show how to increase productivity by automating repetitive, time-consuming tasks when deploying a SAS program across multiple protocols by integrating UNIX shell scripting into your SAS programming.

Shortly after starting on a new clinical reporting team, we were tasked by our sponsor with producing aggregate reports summarizing hundreds of clinical trials (protocols) into one table. The process used their proprietary reporting software to create a new submission and a separate directory for each clinical trial being included in the final report.

We initially used a simple SAS program to copy the datasets from the area where the completed study report was done into our new submission area.

This is done by:

- Copying the program into the first protocol’s directory
- Editing the source data directory LIBNAME statement
- Editing the current protocols data directory LIBNAME statement
- Saving the program.

- Checking the program into the Revision Control System (RCS)
- Executing it from the UNIX command line
- Checking the log for errors
- Executing this series of steps for every protocol in the submission

This copy/ modify/ run/ check action takes about 3-5 minutes to complete, but in a submission incorporating 500 clinical trials, that translates into 1500 to 2500 minutes or 25 to 41 hours of work just to get the data from one place to another. At this rate, we might not make the deadline for this submission!

My solution to this repetitive, time-consuming task is to embed the SAS program in a UNIX shell script, feed it an input file of protocols and source data locations and allow the shell script to dynamically update each LIBNAME statement as each customized SAS program is written out to each protocol's data directory. The shell script then executes each customized program and checks each log file for errors. Thus, the deployDataCopy.ksh utility was born, and 25 to 41 hours of work now gets done in about 15 minutes.

The program has been simplified for the purposes of this paper. I know many SAS programmers may say that all of this can be accomplished using SAS, and there is no need to mix SAS and UNIX programs. I, unfortunately am not that savvy a SAS programmer so this is my solution to the problem. Now let's get to the code.

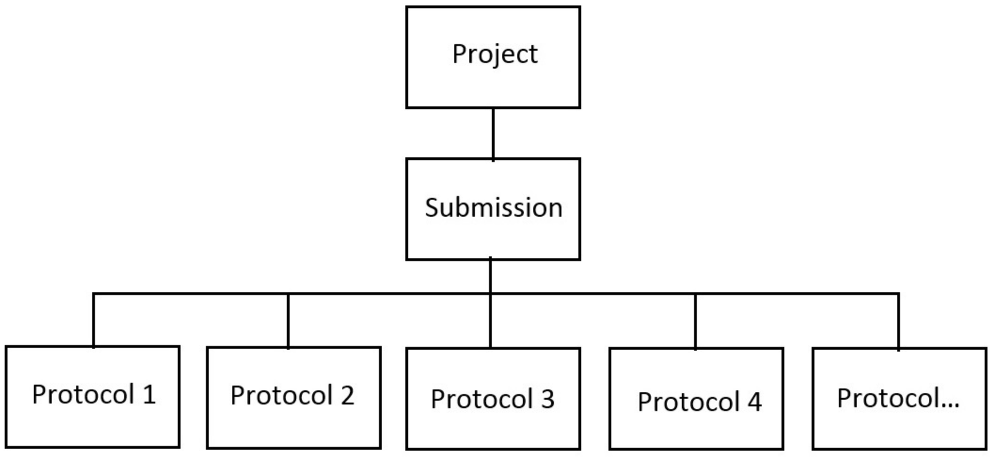
Overview of the file system –

The hierarchy of the structure is illustrated below. Basically the path will look like this:

```
/Volumes/app/data/prod/<project>/<submission>/<protocol>/data
```

For this demonstration, we will be producing a Periodic Benefit-Risk Evaluation Report or as it's known around here a PBRER (pronounced "pē-ber").

Submission Directory Structure



The first part of the code is the user input section. This is where the programmer supplies the project, submission and input filename containing the list of protocols:

```
proj=X335
```

```
sub=pbrer2017

input_file=protocols.lst
```

At this point, our UNIX path looks like this:

```
    /Volumes/app/data/prod/prjX335/pbrer2017/<protocol>/data
```

The protocol variable will be iteratively populated by the contents of the input file

Here is a sample input file, trimmed down to 5 lines for the purposes of this illustration. (We typically have larger files to facilitate the loading of data for hundreds of studies at once). Each line consists of two parts: the protocol name and the path to the source datasets, delimited by a caret:

```
   X3351001^/Volumes/app/data/prod/X335/nda1/X3351001/data
   X3351002^/Volumes/app/data/prod/X335/nda_final/X3351002/data
   X3351007^/Volumes/app/data/prod/X335/csr1/X3351007/data
   X3351012^/Volumes/app/data/prod/X335/csr2/X3351012/data
   X3351015^/Volumes/app/data/prod/X335/csr_pk/X3351015/data
```

The UNIX *cat* function will read in the input file and *pipe* it into a while loop, causing the program to iterate through the input file, one line at a time, until it reaches the end-of-file marker. Each line will be assigned to the variable “${line}”:

```
   cat ./${input_file} |
     while read line
     do
```

Each line can be parsed by using the *cut* command. The first part (protocol name) of the line will be assigned to the variable “${prot}” and the second part (source path) to the variable “${s_path}”:

```
    prot=`echo ${line} | cut -d'^' -f1`
    s_path=`echo ${line} | cut -d'^' -f2`
```

Here are the variable values so far:

Variable	Variable's Value at this step of the Program
${proj}	X335
${sub}	pbrer2017
${input_file}	protocols.lst
${line}	X3351001^/Volumes/app/data/prod/X335/nda1/X3351001/data

${prot}	X3351001
${s_path}	/Volumes/app/data/prod/X335/nda1/X3351001/data

The next two paths are the target directories for the data being loaded and the SAS code that will be executed.

```
d_path=/Volumes/app/data/prod/prj${proj}/${sub}/${prot}/data

p_path=/Volumes/app/data/prod/prj${proj}/${sub}/${prot}/program
```

At this point, all of the program's variables have been set for this iteration of the loop. It is now time to move to the program directory and write out the SAS program. We are using the cat command again, only this time we will create a file in the program directory called DataCopy.sas. The cat command will write everything in this program between the two instances of the word "FINAL" highlighted in yellow below:

```
  cd ${p_path}

    cat > DataCopy.sas << FINAL

/*****************************************************************************
Program name:  DataCopy.sas
Date:          April 24, 2006
Objective:     Copy datasets from location A to location B (version 8).
               Check in & Lock datasets.
               Save the log automatically to program directory.
Instructions:
              - Set up the macro variables for inpath, outpath.
Modifications:
*****************************************************************************/
options mlogic macrogen symbolgen nofmterr nocenter validvarname=any;

libname fmtdata '/Volumes/app/data/prod/formats/v9' access=readonly;

options fmtsearch=(fmtdata.formats);

******************************************
** Set the location for source & target
****************************************** ;
 %let inpath  = ${s_path};
 %let outpath = ${d_path};

******************************************
** Set the dataset flags
****************************************** ;
%let demog=${de_flag};
%let testdrg=${dg_flag};
%let random=${ra_flag};
%macro copydata;
 libname inpath     "&inpath";
 libname outpath v8 "&outpath";
 filename xx pipe "cd &outpath; co -l demog.sas7bdat drug.sas7bdat
random.sas7bdat";

 data _null_;
      infile xx;
      input;
      put _infile_;
 run;

%if &demog=YES %then %do;
 data outpath.demog;
   set inpath.demog;
 run;
%end;
%else %do;
 %put DEMOG dataset does not exist.;
%end;

%if &testdrg=YES %then %do;
    data outpath.drug;
     set inpath.drug;
    run;
%end;
%else %do;
 %put DRUG dataset does not exist.;
%end;

%if &random=YES %then %do;
    data outpath.random;
      set inpath.random;
    run;
%end;
%else %do;
 %put RANDOM dataset does not exist.;
%end;

%mend copydata;
%copydata;

 filename xx pipe "cd &outpath;  ci -u -t-'&subtype data' -m'&subtype data
check-in' *.sas7bdat";

 data _null_;
      infile xx;
      input;
      put _infile_;
 run;


     FINAL
```

The key advantage to embedding the SAS program inside the UNIX shell script is that the UNIX variables can be referenced in the SAS program and will resolve to their UNIX variable values when the SAS program is written out to a file:

```
*****************************************
** Set the location for source & target
***************************************** ;
 %let inpath =${s_path};
 %let outpath=${d_path};

*****************************************
** Set the dataset flags
***************************************** ;
%let demog  = ${de_flag};
%let drug   = ${dg_flag};
%let random = ${ra_flag};
```

As a result, the variable names as they appear in the shell script (above) will be the resolved variable values in the written SAS program (below):

```
*****************************************
** Set the location for source & target
***************************************** ;
 %let inpath =/Volumes/app/data/prod/X335/nda1/X3351001/data;
 %let outpath=/Volumes/app/data/prod/prjX335/pbrer2017/X3351001/data;

*****************************************
** Set the dataset flags
***************************************** ;
%let demog  = YES;
%let drug   = YES;
%let random = YES;
```

Now we are ready to execute the first iteration of the SAS program.

```
sas94 -log ./DataCopy.log -print ./DataCopy.lst -nodms ./DataCopy.sas
done  # End of while loop
```

The *done* statement denotes the end of the *do-while* loop. The iterative process will continue as long as there are lines in the input file containing data loads to be executed.

When all iterations have finished and all source data has been copied to respective target directories, one task remains: checking all of the SAS logs for errors.

By using *grep –i* (ignore case) and a wildcard (*) for the protocol name, the program will check all of the SAS logs at once and write the error messages to a file called log.txt. If the log.txt file has a size greater than 0 *(-s)*, it is written to the screen so the programmer can see which operations resulted in errors. If the size is 0 then the message “No ERRORS found” is output to the screen. Finally, one last bit of housekeeping: remove the log.txt file (if it exists) and exit the program normally.

PART ONE - CONCLUSION

So there you have it: A SAS program embedded in a UNIX shell script, deployed across any number of directories, executed, and then checked for errors. What makes this combination more functional than running either a shell script or SAS program separately, is that embedding the SAS program inside the UNIX shell script allows you the use of UNIX variables in the SAS program. These variables will resolve to their UNIX variable values when the SAS program is stored prior to execution. The man-hours this one utility has saved my team is almost inconceivable due to its ability to dynamically update the SAS program with each iteration of the input file loop thus makes this combination especially useful.

The time savings is very easy to calculate for this illustration. Performing the tasks manually takes 25 to 40 hours of work, whereas setting up and running this shell script takes only 15 minutes. The utility may take an hour to run, but the user need only start the program and move on to other tasks.

In Part 2 of this series I will show how to deploy a standalone SAS program across the same 500 protocols; however, in this example, the program will create its own input file list and then determine on a protocol-by-protocol basis if it needs to be deployed or not.

PART TWO - DEPLOYING A STANDALONE SAS PROGRAM ACROSS MULTIPLE PROTOCOLS USING A UNIX SHELL SCRIPT

There are many other ways to use UNIX shell scripting and SAS® together to increase productivity. Here in part two I will demonstrate the benefits of using a UNIX shell script to deploy a standalone SAS program across any number of protocols. The script keeps protocol-level listings up to date across an entire submission.

This is done by:

- Copying the program into the selected protocol’s data directory
- Checking the program into the Revision Control System (RCS)
- Executing it from the UNIX command line
- Checking the log for errors
- Executing this series of steps for every protocol in the submission

This copy/ run/ check action takes about a minute to complete, but in a submission incorporating 500 clinical trials, that translates into 500 minutes or a little over 8 hours of work. Because the program resides in the same directory as the dataset it is analyzing, there are no LIBNAME statements to update.

This means the program doesn’t have to be embedded in the UNIX shell script, and can instead be deployed as a standalone program using only the shell script to quickly place it where it needs to be. Hence the deployTreatments.ksh utility was born.

Since the utility is frequently run multiple times during the course of a submission, we have added code to dynamically check timestamps and only regenerate listings where the underlying data has changed. The program has been simplified for the purposes of this paper.

This section of code creates an input file of all the treatment datasets in the submission. This file is then piped into a *grep* that removes all the *symlinks* before writing it out to the deploy1.tmp temporary file. The *sed* statement will turn any number of spaces into a single space and allow us to use a space delimiter when the file is read a little further down in the code:

```
ls -l /Volumes/app/data/prod/prj${proj}/${sub}/*/data/treatment.sas7bdat |
grep ^- > ./deploy1.tmp

sed 's/[ ]\{1,\}/ /g' ./deploy1.tmp > ./deploy.lst
```

The program will now use the *cat* command and *while* loop to process the file one line at a time until reaching the end. The first variable assignment is going to remove the entire path and file name from the *ls-l* and store that value in ${line2}. Next, the protocol name will be *cut* from the ${line2} and assigned to

${prot}. Lastly, the *dirname* statement will remove the filename from ${line2}, leaving just the data directory path assigned to ${path}.

Next the program will determine if the Listing from the previous deployment is still “current”. It would be wasteful to deploy a fresh program in each study where the underlying data hasn’t even changed. This is accomplished by doing an *ls -1 –t* into a temporary file, which will produce a reverse chronological listing of the directory contents. Reading this list will allow us to set a flag telling us whether the listing is out-of-date and only regenerate the listing when the underlying data has actually changed:

```
#####################################################################
#  Create directory listing and determine if deployment is needed.
#####################################################################

ls -1 -t > ./dir_lst.txt

flag=no
cat ./dir_lst.txt |
  while read line1
  do

  if [[ ${flag} = "yes" ]]; then
    continue
  elif [[ ${line1} = "treatment.sas7bdat" ]]; then
    echo -e "\n${file_name} is either out of date or does not
exist....deploying"

#####################################################################
#  Check for and remove any existing log, listing and SAS® files.
#####################################################################

  [[ -e ./${file_name}.lst ]] && rm ./${file_name}.lst
  [[ -e ./${file_name}.log ]] && rm ./${file_name}.log
  [[ -e ./${file_name}.sas ]] && rm ./${file_name}.sas

#####################################################################
#  Copy the program to the target directory and execute SAS® program.
#####################################################################

  cp ${src_file} ${file_name}.sas

  echo -e "Running ${file_name}.sas for protocol ${prot}"
    sas94 -log ./${file_name}.log -print ./${file_name}.lst -nodms
./${file_name}.sas

  flag=yes
  elif [[ ${line1} = "${file_name}.lst" ]]; then
    echo -e "\n${file_name} listing is current for protocol ${prot}"
    flag=yes
  fi
  done
done
```

PART TWO - CONCLUSION

As you can see, it is just as simple to deploy a standalone SAS® program using a UNIX shell script as it is to embed one in the program. The two advantages to using the shell script are:

- Programmer setup time is reduced since the utility creates its own input file.
- Total run-time is reduced since the program only regenerates listings which are out-of-date.

The time savings is very easy to calculate for this illustration: Performing the tasks manually takes 8 hours of work versus a 15 second setup time for the shell script. The utility may take an hour to run, but the user need only start the program and move on to other tasks.

PART THREE - DYNAMICALLY WRITING A SAS PROGRAM USING A UNIX SHELL SCRIPT

Clinical trial data doesn't always come in SAS datasets, it can come in a wide range of forms from delimited text files to excel spreadsheets to SAS export files. In all of these cases, these raw data files need to be turned into SAS datasets. This program will dynamically write out the LIBNAME statements for each export file in the directory so the SAS program can iterate through them and unpack each export file.

Usually the first part of the code is the user input section. Since this program is specific to export files and all of the processing happens in the current directory we don't need to supply the program with any information, just put the program into the directory and execute it.

This line of code creates an input file of all xpt files in the current directory:

```
# Create the input list of all export files

ls -1 ./*.xpt > ./setup.lst
```

This is what the current directory looks like before the program is executed. Usually there are dozens of files to be converted, but for our purposes here we will limit the list to 5:

```
    /Volumes/app/data/prod/prjX335/pbrer2017/X3351001/view:
    total 1204
  drwxrwsrwx  2 curlej gaprs   4096 Aug 22 14:34 .
  drwxrwsr-x 15 curlej gaprs  69632 Aug 22 12:55 ..
  -rw-r--r--  1 curlej gaprs 449196 Jul  6 01:40 sas_ADVE.xpt
  -rw-r--r--  1 curlej gaprs  50140 Jul  6 01:40 sas_DEMG.xpt
  -rw-r--r--  1 curlej gaprs 493710 Jul  6 01:40 sas_DRGR.xpt
  -rw-r--r--  1 curlej gaprs  24448 Jul  6 01:41 sas_RAND.xpt
  -rw-r--r--  1 curlej gaprs 123288 Jul  6 01:41 sas_SBSM.xpt
  -rwxrwxrwx  1 curlej gaprs   2774 Aug 22 13:32 unpack_xpt.ksh
```

This step in the program is to iterate through the setup file it just created and write out to a temp file, a LIBNAME statement for each xpt file on the list. The loop counter will iterate 1 more time than we need so the ((x=x-1)) line will bring the counter back to the correct number of LIBNAME statements:

```
cat ./setup.lst |
  while read line
  do

    echo -e "libname xpt${x} xport \"${line}\";" >> ./temp.txt
    ((x=x+1))

done

# The loop counter will increment 1 more than we need so here we
subtract 1
((x=x-1))
```

The contents of the temp.txt file after iterating through the input file:

```
  libname xpt1 xport "./sas_ADVE.xpt";
  libname xpt2 xport "./sas_DEMG.xpt";
  libname xpt3 xport "./sas_DRGR.xpt";
  libname xpt4 xport "./sas_RAND.xpt";
  libname xpt5 xport "./sas_SBSM.xpt";
```

cat will write out the first part of the SAS program to the current directory:

```
# ----------------------------------------------------------------------
# Write out the SAS program to the current directory.
# ----------------------------------------------------------------------

cat > unpack_xpt.sas << FINAL
/**********************************************************************
Program Name:  unpack_xpt.sas
Date:          July 20, 2015
Programmer:    Jim Curley
Description:   Program will unpack a directory full of xpt files.
**********************************************************************/

 options validvarname=v7
         nocenter nodate
         dkricond=warn
         nofmterr nonumber
         macrogen
         mprint mlogic symbolgen;

FINAL
```

The program will now use *cat* again to write out the LIBNAME statements in the temp file to the top of the SAS program. If the temp file does not exist the program will throw an error to the screen, delete all working files and ABEND:

```
   if [[ -e ./temp.txt ]]; then
     cat temp.txt >> unpack_xpt.sas
   else
     echo "ERROR: No XPT files to unpack!!"
     [[ -e ./temp.txt ]] && rm ./temp.txt
     [[ -e ./unpack_xpt.sas ]] && rm ./unpack_xpt.sas
     exit 1
   fi
```

Use *cat* again to write out the rest of the SAS program. Here we see the UNIX script writing out LIBNAME statements and keeping track of the loop counter. Then when writing out this part of the program the UNIX variable ${x} is replaced by the UNIX value so the SAS loop can iterate the same number of times reading the LIBNAME statements as the UNIX loop did writing them:

```
    cat >> unpack_xpt.sas << FINAL1

    libname target "./";

    %macro unpack_it;

      %do i = 1 %to ${x};

         proc copy in=xpt&i out=target;
         run;

      %end;

    proc contents data=target._ALL_;
    run;

    %mend unpack_it;

    %unpack_it;

    FINAL1
```

The rest of the program is just a bit of housekeeping, removing the temp file and input file and executing the SAS program that will convert all of the xpt files into SAS datasets. Then exit the program normally:

```
sas94 -log ./unpack_xpt.log -print ./unpack_xpt.lst -nodms ./unpack_xpt.sas

   exit 0
```

The contents of the directory after the program is executed. There is the original shell script and xpt files plus the dynamically created customized SAS program its log and listing and the newly created SAS datasets.

```
 /Volumes/app/data/prod/prjX335/pbrer2017/X3351001/view:
    total 2492
  drwxrwsrwx  2 curlej gaprs   4096 Aug 22 14:34 .
  drwxrwsr-x 15 curlej gaprs  69632 Aug 22 12:55 ..
  -rw-r--r--  1 curlej gaprs 466944 Aug 22 14:34 adve.sas7bdat
  -rw-r--r--  1 curlej gaprs  73728 Aug 22 14:34 demg.sas7bdat
  -rw-r--r--  1 curlej gaprs 532480 Aug 22 14:34 drgr.sas7bdat
  -rw-r--r--  1 curlej gaprs  73728 Aug 22 14:34 rand.sas7bdat
  -rw-r--r--  1 curlej gaprs 449196 Jul  6 01:40 sas_ADVE.xpt
  -rw-r--r--  1 curlej gaprs  50140 Jul  6 01:40 sas_DEMG.xpt
  -rw-r--r--  1 curlej gaprs 493710 Jul  6 01:40 sas_DRGR.xpt
  -rw-r--r--  1 curlej gaprs  24448 Jul  6 01:41 sas_RAND.xpt
  -rw-r--r--  1 curlej gaprs 123288 Jul  6 01:41 sas_SBSM.xpt
  -rw-r--r--  1 curlej gaprs 139264 Aug 22 14:34 sbsm.sas7bdat
```

```
-rwxrwxrwx  1 curlej gaprs   2774 Aug 22 13:32 unpack_xpt.ksh
-rw-r--r--  1 curlej gaprs   7710 Aug 22 14:34 unpack_xpt.log
-rw-r--r--  1 curlej gaprs  39103 Aug 22 14:34 unpack_xpt.lst
-rw-r--r--  1 curlej gaprs    847 Aug 22 14:34 unpack_xpt.sas
```

PART THREE - CONCLUSION

Here we see how UNIX shell scripts can write customized SAS programs with information gathered on its own. This program takes just a few seconds to get setup and run, saving the programmer the time of manually writing out LIBNAME statements for each file in the directory.

This three part series demonstrates a few different ways that UNIX shell scripting and SAS programming can be used together to dramatically increase the productivity of SAS programmers by automating time-consuming, repetitive tasks.

GLOSSARY OF COMMON UNIX TERMS USED IN THIS PAPER

UNIX Command	Definition
grep	Text search.
cat	For the purposes of this paper cat is used to read input files and to write out embedded SAS programs to a file.
ls	List the contents of a directory.
| (pipe)	Is used to string two or more UNIX commands together. So the output of one command becomes the input for the next command.
sed	Find and replace text strings.
echo	Outputs a message to the screen or used with >> to output to a file.
rm	Remove file. Used to delete temporary files after use.
cut	Used to break up a line at a delimiter.
cp	Copy command.
cd	Change directory.
ci	Check a file into the Revision Control System (RCS).

CONTACT INFORMATION

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

Jim Curley
Eliassen Group
JCurley@Eliassen.com
http://www.jimcurley.net/sas-global-forum-2018