

An Introduction to UNIX Shell Programming and the SAS System

Gregory S. Nelson

Glenn Leavell

The University of Georgia

SAS is a registered trademark or trademark of SAS Institute Inc.
in the USA and other countries.

Other brand and product names are registered trademarks or
trademarks of their respective companies.

Abstract

UNIX is a powerful operating system which has developed into one of the most versatile computing environments. By virtue of its power, flexibility and openness, the UNIX operating system and its variants have at their disposal a variety of advantages for the SAS programmer/ developer. This paper will focus on one particular aspect of the UNIX environment which enables SAS programs to capture information passed along by the operating system: the UNIX shell.

The goal of this paper is not to teach you UNIX or how to write sophisticated shell programs. Instead, we aim to give you a general understanding of shells, shell programming, and other shell concepts, and to show you how you can take advantage of them when using SAS in a UNIX environment. Part I of this paper deals exclusively with shells and shell concepts. Part II talks specifically about how you can use those concepts when programming in SAS.

UNIX Shells

Simply put, a UNIX shell is a program that accepts and interprets commands and then has the operating system execute them. Shells are used for an endless number of tasks, and if you've used UNIX before, then you've used a shell, whether you realize it or not. Shells can be used in both interactive and batch mode, and they take on different personalities in each mode. In this part of the paper, we'll discuss shells at length and tell you how you can use them to enhance your UNIX SAS programs. We'll start with a basic introduction to shells and then move to

some more advanced shell concepts, including shell programming, subshells, and shell variables.

A Shell is a Command Interpreter

A shell can be loosely defined as any program in UNIX whose function is to interpret and execute commands. A shell can carry out this function in either batch or interactive modes. In interactive mode, a shell will most often provide you with a prompt where you can type a variety of commands. When you first login to a UNIX system, you are typically presented with such a prompt, unless you are using a Graphical User Interface, such as the X-Windows system, that has been configured to hide this prompt from you. In batch mode, shells are used to create programs made up of specific commands. These programs are called "shell scripts".

The Login Shell

As we stated before, a shell is a command interpreter. A command interpreter is a program designed to interpret and then help execute your commands. When you login interactively to a UNIX session, you will, of course, want to execute some commands. So, every account on a UNIX system is designated to automatically start a specific program when the user of that account logs in. This program is generically referred to as the user's *login shell*. Typically, this shell is a program that presents the user with some sort of command prompt. Because the look of the command prompt can be easily customized, its look can vary greatly from

machine to machine, but if you were to login to a machine named "groucho", the command prompt might look like this:

```
groucho%
```

Some UNIX accounts have been customized either by the owner of the account or by the system administrator of the machine so that after the login shell first starts it will automatically start some sort of Graphical User Interface (GUI) like X-Windows, when the user initially logs in. These accounts may not immediately present a command prompt, but the owner can easily start some sort of command window in order to get access to a command prompt.

At the command prompt, you could type any number of commands like "ls", "cp", "mv", "sas", "mail", etc. When you do type in a command, the shell will act as a *command interpreter*. This means that the shell will evaluate your command, and then pass it on to the operating system for execution.

During your entire UNIX login session, this initial shell program is always running. In fact, your login session is terminated by definition when your login shell ends or exits. The command to end most login shells is probably familiar to you: "logout".

To summarize, when you login to an interactive UNIX session, your login shell automatically runs, presenting you with a command prompt, accepting commands, and acting as a command interpreter. When this shell program ends, so does your login session.

Specific Shells

So far, we've been talking about shells in generic terms. Specifically, though, there are a wide variety of shells from which to choose. Each has its own list of features. Examples of some such features are the ability to scroll through an edit a set of previously executed commands and the ability to create customized commands known as "aliases". Some of the most popular shells are the C-shell (csh), the Bourne Shell (sh), the Korn Shell (ksh), the Tcsh (tcsh), and the Bourne Again Shell (bash).

The Bourne shell is essentially the original shell. Although it is rarely used for interactive work today, it is still used regularly for shell programming. The C-shell, developed at Berkeley, was designed with interactive use in mind. It is the standard shell on very many

UNIX systems, and it can be argued that it is the most popular interactive shell in use today. The Korn Shell is the standard shell under System V, Release 4 UNIX. It is a superset of the Bourne Shell, providing many enhanced features. The Tcsh, available free from Cornell, is a very popular enhanced version of the C-shell. Bash, a derivative of the Bourne shell, is the shell offered by the Free Software Foundation. It is also the standard shell on the Linux operating system. *Basically, almost all shells fall into one of two classes: those derived from the Bourne shell and those derived from the C-shell.*

Each of these shells share similarities, but they are all different. So, commands that work on one UNIX account may not work on another account on the same system; it all depends on which shell you are using. On many systems, the user can choose to change their default shell at any time. The commands "chsh" (Change Shell) and "passwd -s" are often used for this procedure. Once you change your default shell, you'll have to log out and log back in to start using the new shell.

Local Commands and Internal Commands

Most of the commands that you type at the shell prompt, like "ls", "mv", "sas", etc., are actually programs that reside on the system on which you are working. When you type commands like these, the shell instructs the operating system to load the command from disk and to run it. However each shell also comes with a set of its own internal commands. Each shell's own set of internal shell commands is what makes each shell different from the rest. For example, if you are using the C-shell, and you type the command "limit", you'll get a list of various resource limits that have been specified for your account. However, the "limit" command does not exist in the Korn or Bourne Shells. If you were to type the command in one of those shells, you'd get a *command not found* type of error.

Shell Scripts

We've been talking about the use of shell programs as interactive command interpreters. Shells can also be used in a batch mode in a form called a "shell script". A shell script is simply a list of commands organized together to create a program. To help organize these commands, each shell also has its own set of language commands, or programming construct

syntax. The simplest type of shell script does not contain any of these programming constructs, but consists only of several single commands, each on a line. For example, the following shell script renames the file *job3.sas* to *current.sas*, runs the sas job *current.sas*, and then prints the current date and time:

```
mv job3.sas current.sas
sas current.sas
date
```

A more complicated script might first to check to make sure that the file *job3.sas* exists before doing anything else. If you were using the Bourne shell or one of its derivatives, an example would be:

```
if [ -f job3.sas ]; then
    mv job3.sas current.sas
    sas current.sas
    date
else
    echo job3.sas is not found
fi
```

Actually Running a Shell Script

At this point, we've seen a couple of shell script examples, but we haven't discussed exactly how to get those scripts to run. Let's assume that you wanted to run the shell script in the previous example. The first step is to type the script into a file. There are lots of ways to do this, but using your favorite editor is probably the easiest. Assume that you called the file *myscript*. After saving the file, the next step is to mark the file as being *executable*. To do this, type the UNIX command:

```
chmod +x myscript
```

You can now run the script by typing the command:

```
./myscript
```

The *./* prefix indicates that the file is located in the current directory. It may or may not be necessary to use this prefix, depending on how your account is configured.

When you run a script like this, the commands are executed using an instance of the same type of shell that you happen to be using as your interactive or login shell. Remember that we said that the above script was written to be run

under the Bourne shell or one of its derivatives. That means that the script would work just fine if we happened to be using one of those shells, but that it would produce errors if you were using a different type of shell, like the C-shell. So that shell scripts don't have to be dependent on the type of shell being used when they are called, a method exists for you to specify a specific shell to be used with your shells scripts. To make this specification, simply add a new line to the top of the script. This line will consist of the characters *#!/* followed by the full path name of the shell that we wish to use. In UNIX, most shells are located in the */bin* subdirectory. So, if we wanted our second example to run under the Bourne shell, it would become:

```
#!/bin/sh
if [ -f job3.sas ]; then
    mv job3.sas current.sas
    sas current.sas
    date
else
    echo job3.sas is not found
fi
```

That first line will force the program to run under the Bourne shell, so you won't get any errors when you run it, no matter which shell you happen to be using interactively.

Subshells

When you run a shell script, the operating system actually starts up a brand new instance of the shell necessary to run the script. For example, if you were using the C-shell interactively, and you ran our Bourne shell example from the previous section, the operating system would start running a brand new Bourne shell (the */bin/sh* program) instance. This shell would continue to run until it had finished executing the script.

Whenever a program starts a new instance of a shell, that new shell is called a *subshell*. Running shell scripts is one of the most common ways that subshells are created. However, you can also create a subshell interactively. For example, suppose you are using the C-shell as your login shell, but that you wanted to start a Bourne subshell so that you could type some commands that were only available in the Bourne shell. To do this, you could simply type the command:

```
/bin/sh
```

at the command line. After doing this, you might get a different command prompt, depending on how your account and system have been configured. Although your environment may appear similar after typing this command, it's definitely not the same as it was before typing the command. Before, you were using the C-shell. Now, you're using the Bourne shell. The "exit" command can be used to terminate a subshell. So to get *back* to your original shell, simply type the command "exit".

Note that the "logout" command can *not* be used to exit from a subshell. That's because even though the subshell might be interactive, it's not a login shell, and the "logout" command is only used to terminate a login shell.

One thing that makes subshells interesting is that they inherit properties from the programs (usually other shells) that call them. We'll discuss more on that topic in the "Shell and Environment Variables" section below.

"Sourcing" a Script

In most cases, when you run a shell script, you want it to be executed by a brand new shell instance - a subshell. However, there are times when you might want the commands in your script to be executed *by the specific shell instance that you are currently using*. This is commonly called "sourcing" the script. The syntax for sourcing depends on what shell you are using. To source a script in the C-shell or one of its derivatives, type the command "source scriptname". To source a script in the Bourne shell or one of its derivatives, type the command ". scriptname" (note that in this case, the period is used as an actual command). So, if you were using the Bourne shell and wanted to source the *myscript* script that you created earlier, you would type:

```
. myscript
```

Any special "#!" shell definition will be ignored when sourcing a script. Because sourcing a script always runs it under the *current* shell, that script *must* be written for that particular type of shell.

Startup File Explanation

When each shell first starts, it normally processes one or more special shell scripts automatically. You can use these scripts to help customize your general shell scripts and your

account. Which special scripts get run depends on which shell you're using, and on whether or not the shell is a login shell. Here is a table of some common shells, and which scripts get executed automatically when they run.

	csh/tsch	sh	ksh
Always	.cshrc	NONE	.kshrc ¹
Login	.login	.profile	.profile

In order to work, these files should be located in your home directory (the directory where you are located when you first login). When most accounts are created, one or more of these files are usually created and put in your home directory automatically. Your systems administrator may also have some system-wide startup scripts that run whenever a shell is started. If so, the system-wide scripts will be run before the startup scripts in your home directory get run.

Note that all shell startup scripts are *sourced*, meaning that a new subshell is *not* created when they run. Instead, the commands in the startup scripts are executed by the shell that is starting.

So, assume that your login shell the C-shell. When you first login, the .cshrc script will run, as it always does when an instance of /bin/csh starts. Then, the .login script will run, because it's a login shell. If you were to type the command "/bin/csh" to start a new C-shell subshell, then only the ".cshrc" script would run.

Although the topic will not be discussed in this paper, it is possible to pass special flags to a script to prevent it from running startup scripts.

Shell and Environment Variables

When working and programming with shells, it is often useful to create and set what are known as "shell variables". While almost any shell will support the use of shell variables, the syntax for defining and using those variables varies from shell to shell.

¹ The .kshrc file will not run unless the ENV environment variable is set to "\$HOME/.kshrc". This is usually done in .profile. The value of ENV can really be anything; .kshrc is a standard convention. See the "Shell and Environment Variables" section for a general discussion of environment variables.

Creating a shell variable is easy. In the C-shell and its derivatives, the command is:

```
set variable-name=value
```

In the Bourne shell and its derivatives, the command is:

```
variable-name=value
```

If you were using the C-shell, you could type the following command to create a variable called *fruit* and assign it the value *apple*:

```
set fruit=apple
```

The command in the Bourne shell would be:

```
fruit=apple
```

Shell variables are referenced by placing a dollar sign (\$) in front of their name, so to print the value of the variable *fruit*, you could type:

```
echo $fruit
```

This command would work no matter what shell you were using.

You can create and use shell variables both interactively and within shell scripts. Note that shell variables, in their simplest form, only exist within the shell or subshell that created them. Once that shell ends, the variable definition vanishes. Likewise, if one shell starts a subshell, the shell variables from the first shell will not be visible to the subshell.

Because you often *will* want to pass variable information from one shell to a subshell, a special type of shell variable called an *environment variable* exists to make this possible. If you create an environment variable within a shell, that variable *can* be seen and referenced by any subshell created by that shell. Note, however, that any changes made to the environment variable *within* the subshell will *not* effect the value of the variable as it stands within in shell that called the subshell.

In the C-shell and its derivatives, you define an environment variable by using the command:

```
setenv variable-name value
```

So, to create a variable called *fruit* in the C-shell and assign it the value *apple*, the command would be:

```
setenv fruit apple
```

Note that unlike the "set" command, no equal sign ("=") is used with the "setenv" command.

The method for creating an environment variable in the Bourne shell and its derivatives is a little different. In these shells, you first create a standard shell variable and then *convert* that variable to an environment variable. This process of conversion is called "exporting" the variable. The command for exporting a variable is simply:

```
export variable-name
```

So, the full set of commands to create an environment variable² in the Bourne shell called *fruit* with the value *apple* is:

```
fruit=apple
export fruit
```

Keep in mind that shell variables, like most things in UNIX, are case-sensitive. This means that the variable *fruit* and the variable *FRUIT* are two separate variables. Also, it's a convention to make standard shell variables all lower-case, and to make environment variables all upper-case. Note that we didn't follow this convention in the above examples, to avoid confusion.

Listing Shell Variables

Each shell provides commands that you can use to list all of your shell and environment variables. In both C-shell and Bourne shell derivatives the command "set" will list your shell variables. In the C-shell, the list will contain only the shell's standard shell variables, *not* the environment variables. In the Bourne shell, the "set" command will list all shell variables, *including* environment variables. You can use the command "env" in all of the above shells to get a list of environment variables only. In the C-shell, the command "setenv" without any arguments will also work.

² The Korn shell provides a "shortcut" for creating environment variables, in that the "export" command can be used to both set the value of and export the variable. The syntax of the shortcut is "export variable-name=value".

Standard Environment Variables

There are several standard environment variables that the shells will automatically set for you. They're often useful when writing shell scripts. Some of the more useful ones are:

Environment Variable	Description
TERM	the current terminal type in use
USER	the login name of the current user
HOME	the path name of the user's home directory
PWD	the "present working directory" (current directory)
HOST	the name of the UNIX host that you are using
\$	the process id of the current shell

So, the command "echo \$USER" will print out your user name and the command "echo \$\$" will print out the UNIX process id of your current shell.

Using UNIX Shell Features with the SAS System

There are a variety of methods that the SAS programmer can employ to interact with the UNIX environment, both in terms of executing commands as well as capturing shell and environment variables in your programs. Depending upon your application you have the following at your disposal:

- The X Statement
- The %SYSEXEC Macro
- The CALL SYSTEM Routine
- The PIPE Device Type
- The %SYSGET macro and SYSGET Function

In John Sims *Observations* article (see References), he deals specifically with how to pass information back and forth between SAS and programs running in the UNIX environment.

Here we focus more specifically on how to get shell and environment variables into your SAS programs. Although we will briefly discuss executing UNIX commands from within SAS, please refer to Sims, 1995 for more information.

Executing UNIX Commands from within the SAS System

The X Statement

The X statement in SAS allows the program to execute a operating system specific command from within a SAS program. With this command, you can execute a single UNIX command or start a shell script. At the conclusion of the X command you have at your disposal a macro variable (%SYSRC) which contains the return code of the last operating system command you ran. The X statement can be useful when you want to list all the available processes, change directories, set environment variables etc. However, note that three UNIX commands are built in to SAS: "cd", "pwd" and "setenv", because they relate directly with the SAS session environment.

Suppose in our SAS program, we wanted to change directories so we could include some code from the user's home or login directory. We can either use the ~ (tilde) or the environment variable \$HOME to make that change.

```
x cd ~/SASDATA ;
or
x cd $HOME/SASDATA;
x pwd ;
run;
```

This program will produce something similar to the following in our SAS Log:

NOTE: Current working directory is 'bl/staff/gnelson/sesug'.

The %SYSEXEC Macro

The %SYSEXEC macro program statement essentially provides the same functionality as the X statement and can only be used in open code and not in a DATA step. The %SYSEXEC macro statement executes a system command immediately. Using the previous example, here we use the %SYSEXEC macro:

```
%SYSEXEC cd $HOME/SASDATA ;
```

```
%SYSEXEC pwd ;
```

And the result to our SAS Log is identical.

NOTE: Current working directory is 'b1/staff/gnelson/sesug'.

The CALL SYSTEM Routine

The CALL SYSTEM routine provides a vehicle for issuing operating system commands from within a DATA step.

Depending on whether you are running SAS in an X-Windows environment or in a character-based environment, the output of the command will either appear in the window from which you invoke the SAS System or clear the window, display the output and prompt you to continue. An example of the CALL SYSTEM routine within a DATA step might be displaying a list of all the processes (jobs) I have running. A system command can take on these various forms:

A UNIX command enclosed in quotes

```
Data _null_;
  call system('ps aux | grep
$USER ');
run;
```

An expression whose value is a UNIX command

```
Data _null_;
  process = "ps aux";
  pipen = "|";
  whoseon = "grep $USER";
  call system(process || pipen ||
whoseon);
run;
```

The name of a character variable whose value is a UNIX command

```
Data _null_;
  process = "ps aux | grep $USER";
  call system(process);
run;
```

The PIPE Device Type

For those interested in reading data directly from a UNIX process or writing data to another process, the PIPE device type on the FILENAME statement can be useful. For example, if I wanted to mail myself a message after a SAS program ran to tell me whether or not I had any errors, I might use the PIPE device type.

```
Filename mailer pipe 'mail -s
"Weekly Job Status" $USER';
Data _null_;
  set weekly.jobstat;
  file mailer;
  If statusrc='OK' then put
  'Everything ran fine';
  else put 'Something was wrong,
you need to check things.';
run;
```

The %SYSGET macro and SYSGET Function

Most of what we have discussed so far with respect to SAS has been how to get SAS to execute UNIX commands. The commands can be UNIX commands (like "ls", "cd", "pwd") as well as shell scripts that we've written to provide some special functionality. What if we simply want to get the value of some shell or environment variable and put that into a SAS variable or macro variable? The tool(s) that we use here are the %SYSGET macro statement and the SYSGET function.

The %SYSGET function is a routine which allows us to simply capture the value of an environment variable. For example, say I want to know what kind of terminal emulation the user of this program is on, I can use the TERM environment variable.

```
data _null_;
  terminal = sysget('TERM');
  put terminal= ;
run;
```

Would result in:

```
TERMINAL=xterm
```

The %SYSGET macro statement is similar to the SYSGET function except that it applies to macro variables. For example, the following would write the value of the time zone to the SAS Log:

```
%let timezone =%sysget(TZ);
%put &timezone;
```

Examples

Example #1 - Shell Script:

Using a shell script to start the SAS System with multiple AUTOEXEC.SAS files

Occasionally, we need to do things in SAS that aren't automatically built into the SAS System. In the example, below we simply start SAS by

passing it various parameters. Generally what this script does is it looks for three files at startup. These three files contain valid SAS statements to include when SAS starts. Most commonly, we use these to establish libraries, macro options and locations of format libraries. Here we look for the user's personal autoexec.sas file located in his/her home directory (\$HOME). Next we set up the system tools and libraries. Finally, we look in a reserved directory for any libraries that a user has so they don't have to establish those each time they invoke SAS.

```
#!/bin/sh

# This is a front-end script for sas. It configures the TERMINFO and
# TERINFOADD variables to point to the database provided by sas. It
# also makes life easier by not trying to run sas under the X-windows
# environment when the DISPLAY environment is not set. If the DISPLAY
# variable IS set, it let's the user's defaults prevail. User
# specified command line options are also taken into account.

# Gregory S. Nelson - Created 18APR95

#                               Modified 19MAY95 - Multiple Autoexec Files
#                               Modified 26MAY95 - Changed location of
#                               autoexec.sas and
#                               userid.autoexec
#                               Modified 21JUN95 - Modified script to direct
#                               temp file to TFILE

# The next few lines contain all the local changes.
# -----
# This is where sas lives
#SASHOME=/usr/lib/sas609
#SASHOME=/usr/local/bin/

# This points the location of temporary space
#TFILE=/tmp/autotemp.$$

# This points to the user's personal autoexec.sas file
#PERSONAL=$HOME/autoexec.sas

# System wide autoexec file - used for system defaults, macro
# variables, etc.
#SYSTEM=$HOME/project/tools/ultrix/sysauto.sas

# Facility creates autoexec file for user based on any libraries
# "registered" in the system.
#FACILITY=$HOME/project/reserved/$USER.autoexec

# These point to the term databases provided with the sas program
#TERMINFO=$SASHOME/terminfo
#TERMINFOADD=$SASHOME/terminfo
# -----
# If the DISPLAY variable is set, then run sas according to the
# prevailing defaults, which are determined by the system or user's
```

```
# sas.config file
```

```
if [ -n "$DISPLAY" ]; then
  ($SASHOME/sas -autoexec `(cat -s $FACILITY $PERSONAL $SYSTEM > $TFILE ;echo $
TFILE)` $* ; rm $TFILE)
else
  ($SASHOME/sas -autoexec `(cat -s $FACILITY $PERSONAL $SYSTEM > $TFILE ;echo $
TFILE)` -fsdevice ascii $* ; rm $TFILE)
fi
```

Example #2 - SAS Programming

Looking at specific directories and "automagically" creating personal libname statements and reports for each user

The next example takes a certain directory and reviews the owner name of the directories located there. Once it has the owner name and name of

the directory, it creates a report of all the libraries within that hierarchy and creates three files: a report containing all the libraries in the system and who their owner is, a SAS program containing all the libname statements for all of the libraries in the system and a file for each user containing just their libraries with valid libname statements.

```
/* * -----
* ENVIRON.SAS
* Purpose: Establishes user environments for the PROJECT
* Data Management Facility.
*
* Invocation:
* Should be run by root or Database Manager - requires
* write access to /project/reserved directory.
*
* Processes
* Involved:
* Searches /project/data hierarchy for directory
* names and checks for validity of dataset name.
*
* Writes file:
*
* SAS Data Sets:
*
* _project_.all_libs: dataset with all of the libraries
* in the system and their owners.
*
* Flat Files:
*
* PROJECTROOT/reserved/library.all - report of
* all of the libraries and their owners.
*
* PROJECTROOT/reserved/libs.all - libname statements
* for all of the libraries in the system.
*
* PROJECTROOT/reserved/$USER.autoexec - libname statements
* for each individual having libraries in the system.
*
* Reference: Gregory S. Nelson 18APR95
* Modified:
* ----- */
/* ----- */
/* ----- Macro Variables for UNIX System ----- */
/* ----- */
%let f_root = ~/project ;
%let da_root = ~/project/data ;
%let r_root = ~/project/reserved ;
%let de_root = ~/project/design ;
%let t_root = ~/project/tools/ultrix ;
%let yy = / ;
%let del = rm ;
%let home = ~ ;
```

```

/* ----- */
/* ----- */
/* Get a directory listing so we can tell which Projects are */
/* in the PROJECT system... found in /project/data/* heirarchy. */
/* ----- */
Filename dirlist "&r_root.&yy.temp2.dir";
/*
Data _null_;
  call system("ls -Fl&a &da_root > &r_root.&yy.temp.dir");
run;
*/

Data _project_.all_libs (drop= flg_1 flg_2);
  infile dirlist lrecl=80 pad firstobs=3;
  input @1 flg_1 $1. @46 flg_2 $1. @; /* Get valid directory entries */

  If flg_1 ne 'd' or flg_2 = ' '
    then delete
  ;
  else /* Read only the fields we care */
    Input @15 owner $12.
          @46 d_name $9.
    ;
    /* Drop the trailing slash */
    /* from new variable */

    d_name=substr(d_name,1,length(trim(d_name))-1);
    f_name=("&r_root.&yy"||substr(owner,1,length(trim(owner))))||".autoexec";

    /* Check to see if the var */
    /* is a valid SAS var name */

    If 'A' <= upcase(d_name) <= 'Z'
      or index(upcase(substr(d_name,1,1)), '_')
    then do;
      ds_name=d_name;
      output;
    end;

    Else If length(d_name) < 8 then do;
      ds_name='_'||d_name;
      output;
    end;

    Else do;
      ds_name='_'||substr(d_name,1,7);
      flg_3=1;
      put "Possible Invalid Library Name= " ds_name;
      output;
    end;

run;

/* ----- */
/* Clean up after ourselves - delete directory listing. */
/* ----- */

* x * &del &r_root.&yy.temp.dir";

/* ----- */
/* Create a report of all studies in the system including */
/* the owner (looking at permission at UNIX Level for Owner. */
/* ----- */

Proc sort data=_project_.all_libs;
  by flg_3 owner;
run;

Data _null_;
  set _project_.all_libs;
  by flg_3 owner;
  file "&r_root.&yy.library.all";
  If _n_=1 then
    Put / @10 "PROJECT Data Management Facility Contents"
        // @10 "Report Generated on &SYSDATE at &SYSTIME"
        // @15 "Owner Userid" @30 "Study Name"
        / @15 "===== " @30 "===== "
        ;
  ;

```

```

        Put      @15 owner @30 ds_name;
run;

/* ----- */
/*      Write the File Libs.all which contains all the valid      */
/*      Libnames for all studies found in the System.              */
/*      Before we write them, check to see if they have been      */
/*      flagged as possible conflicting library entries and        */
/*      separate them from the others...                            */
/* ----- */

Data _null_;
  set _project_.all_libs;
  by flg_3 owner;
  file "&root.&yy.libs.all";

/* ----- */
/*      For those Library Names that I had to truncate, let's put them */
/*      by themselves so we can manually check them...              */
/* ----- */

  if flg_3=1 and first.flg_3 then do;
    Put // " " --- Check for invalid Libnames Below ---;" ;
    Put " " I truncated the Libname, but they may need to be checked;" ;
  end;

  if flg_3=1 then do;
    Put "Libname " ds_name " " "&root.&yy" d_name +(-1) " " ;
  end;

  if flg_3=1 and last.flg_3 then do;
    Put " " --- End of invalid Libnames Above ---;" //;
  end;

  Else do;
    Put "Libname " ds_name " " "&root.&yy" d_name +(-1) " " ;
  end;

run;

/* ----- */
/*      Ok, now let's write all the others in each of their mylibs.sas */
/*      files so that each individual has their own libraries.        */
/* ----- */

Data _null_;
  set _project_.all_libs;
  by flg_3 owner;
  file greg filevar=f_name ;

/* ----- */
/*      For those Library Names that I had to truncate, let's put them */
/*      by themselves so we can manually check them...              */
/* ----- */

  if flg_3=1 and first.flg_3 then do;
    Put // " " --- Check for invalid Libnames Below ---;" ;
    Put " " I truncated the Libname, but they may need to be checked;" ;
  end;

  if flg_3=1 then do;
    Put "Libname " ds_name " " "&root.&yy" d_name +(-1) " " ;
  end;

  if flg_3=1 and last.flg_3 then do;
    Put " " --- End of invalid Libnames Above ---;" //;
  end;

  Else do;
    Put "Libname " ds_name " " "&root.&yy" d_name +(-1) " " ;
  end;

run;

```

Summary

The UNIX operating system provides a rich feature set to the SAS programmer. Understanding the shells and shell concepts will help you understand the capabilities of SAS in this environment. This paper has sought to touch on some of the ways in which the SAS aficionado can get the most of UNIX.

In general, we have talked about the methods by which we obtain the values of shell variables and environment variables and how to use that information within SAS. When you want to run a UNIX command, use the X statement, the %SYSEXEC macro program statement, or the CALL SYSTEM routine. When you want to obtain the value of an environment variable, use the \$\$SYSGET macro program statement or the SYSGET function. Please refer to Sims, 1995 for a more detailed explanation of the advantages and disadvantages of these methods.

References

More in-depth information about shells and shell programming is available from various books and on-line resources. Following are pointers to several.

Books ...

SAS and UNIX Specific References

Nelson³, G.S. (1995) The SAS System UNIX Primer. Invited paper presented at the annual convention of the International SAS Users Group (SUGI), Orlando, FL.

Sims, John (1995), Using SAS Software and Other Programs in the UNIX Environment. Observations, First Quarter, 1995. SAS Institute Inc., Cary, NC.

SAS Companion for the UNIX Environment and Derivatives,

³ The complete Primer can be obtained by requesting it directly from the author. An outline is presented in the SUGI 20 Proceedings.

Version 6, First Edition. SAS Institute Inc., Cary, NC.

SAS Companion for UNIX Environments: User Interfaces, Version 6, First Edition. SAS Institute Inc., Cary, NC.

SAS Companion for UNIX Environments: Language, Version 6, First Edition. SAS Institute Inc., Cary, NC.

UNIX and Shell References

O'Reilly & Associates, Inc., publisher of the well-known "Nutshell Handbooks", offers several books on shells and shell programming. These books can be ordered through your local bookstore, or you can order them from ORA directly. Their online catalog is at <http://www.ora.com> and gopher.ora.com, and you can reach them by phone at 800-998-9938 (US/Canada).

Rosenblatt, Bill (June 1993) *Learning the Korn Shell*, 1st Edition ISBN 1-56592-054-6

DuBois, Paul (August 1995 - Estimated) *Using csh and tcsh* 1st Edition ISBN 1-56592-132-1

Newbam, Cameron & Rosenblatt, Bill (August 1995 - Estimated) *Learning the Bash Shell* 1st Edition ISBN 1-56592-147-X

On-line Resources ...

The Usenet Newsgroups `comp.unix.shells` and `comp.unix.questions` provide a good forum for learning both about shells and UNIX in general.

Section 5 of the document "UNIX - Frequently Asked Questions", written almost entirely by Matthew Wicks <wicks@dcdm.jw.fnal.gov>, deals exclusively with the topic of shells. The document is posted frequently to both the `comp.unix.shells` and `comp.unix.questions` newsgroups. You can also find the section archived at URL:

<ftp://rtfm.mit.edu/pub/usenet/new.s.answers/unix-faq/faq/part5>

The document "UNIX Shell Differences and How to Change Your Shell", written by Brian Blackmore <bnb@gryphon.demon.co.uk>, offers an excellent comparison of shells, including a section on the history of shells and a detailed

table comparing shells and shell functionality. The document is posted monthly to the comp.unix.shells newsgroup. You can also find it archived at URL:

`ftp://rtfm.mit.edu//pub/usenet/unix-faq/shell/shell-differences`

The document "The Ultimate Shell - Conversion / Equivalence Table", written by Ove Ruben R Olsen <Ruben@uib.no>, offers very extensive tables comparing the various shells' parameters, flags, internals, input/output mechanisms, and environment variables. The document is archived at the following URLs:

`ftp://cs.uwp.edu/pub/vi/docs/shell-101.BetaA.Z`

`ftp://utsun.s.u-tokyo.ac.jp/misc/vi-archive/docs/shell-101.BetaA.Z`

`ftp://alf.uib.no/pub/vi/docs/shell-101.BetaA.Z`

The document "Csh Programming Considered Harmful", by Tom Christiansen <tchrist@mox.perl.com>, explains why, in the author's opinion, the C shell is inadequate for programming. The document is archived at URL:

`ftp://rtfm.mit.edu//pub/usenet/unix-faq/shell/csh-whynot`

Address

Gregory S. Nelson
179 Computer Services Annex
The University of Georgia
Athens, GA 30602-1911
gnelson@dcs.uga.edu

Glenn Leavell
Computer Center
Terry College of Business
The University of Georgia
Athens, GA 30602-6270
glenn@cba.uga.edu

Acknowledgements

We would like to acknowledge the assistance of John Sims and Cyndie Gareleck for their assistance in reviewing this manuscript.