Paper 049-2007

SAS Tools for Program Automation

Richard F. Pless, Ovation Research Group, Highland Park, IL

ABSTRACT

Very often programmers find themselves re-running the same code with slight modifications to reflect a new filename, date or directory. How many times have you found yourself making minor edits to existing code to help it run on a new day or handle a slightly different input file? How many times have administrative notes not been updated such as who ran the code or where it is located? Even worse, how many times have deadlines been missed because a program's author wasn't there to make the necessary minor edits to the code? Thankfully SAS has a number of tools to help you reduce the tedium of manual updates and avoid their associated errors.

SAS provides a number of resources that can be easily incorporated in your code to make your programs more extensible and robust. Many hard-coded dataset and variable references in your code can be replaced through the use of SAS dictionaries. Through the use of automatic macro variables, DATA steps and PROCs can be run conditionally based on the proper execution of previous code. Automatic macro variables and information from the operating system can also enhance logs with runtime information.

This paper will introduce the basics of dictionaries, automatic macro variables, and operating system environment variables as well as provide a number of examples of how they might be used to automate code and control program flow.

INTRODUCTION

This paper is intended to provide the reader with an overview of the SAS dictionaries and automatic macro variables, to provide some examples of possible uses, and to show some programming issues that may be encountered while using these features. A minimal understanding of macro variables and PROC SQL is assumed.

PROGRAMMING WHEN YOU DON'T KNOW OBJECT NAMES: AN INTRO TO DICTIONARIES

One of the most challenging aspects of writing re-usable programs is the need to update trivial information before actually re-running the code. Good programmers typically address this problem by formatting the code so that the necessary updates are readily apparent when the code is about to be re-used. Extensive comments and assigning values to macro variables with %LET are the usual tools and they are very effective. They aren't the only possible solutions, however. One major drawback of such solutions is that they require the programmer to know the names of their objects (such as datasets or variables) prior to using them. You can't use %LET to assign a value to a macro variable if you don't know what you should assign.

Fortunately, many problems don't require you to know the name of the tables or variables that you are going to manipulate. Very often you can get around such trivial updates to your code by having the program itself find the names of relevant objects as it is running. This section will introduce you to SAS DICTIONARIES and how to use them to write code when you don't know the name of the objects.

ACCESSING THE DICTIONARIES

SAS DICTIONARY tables contain information about your SAS session including details about SAS options and objects (e.g. tables, catalogs, variables, indexes, etc) available at runtime. DICTIONARIES are available to PROC SQL but not to other PROCs or the DATA step. If you are SQL-averse you can access the same data via views in the SASHELP library that is available at runtime. Both approaches access the same information, but the means of accessing the information differs. SAS documentation indicates that using SQL has some performance advantages that may be important to some users. This paper will use PROC SQL to access the DICTIONARY contents but similar DATA step methods which may be used in conjunction with SASHELP datasets.

CREATE A LIST OF DATASETS MODIFIED TODAY (WITH DICTIONARY.TABLES)

Very often it is useful to see what files were modified recently. The DICTIONARY.TABLES file provides you with information that can help. It contains information about SAS datasets such as creation and modification dates, number of variables and observations, as well as much of the other information normally presented in the header of PROC CONTENTS output.

Suppose that you want to create a list of recently modified datasets for later use in your code. Rather than manually looking up modification dates and altering your code, you can use DICTIONARY.TABLES. The code below finds all of the datasets in the library MYDIR which were modified within the past day and assigns the names to the macro variable NEWDS (multiple values are separated by commas).

```
proc sql noprint;
       select memname
       into :newds separated by ','
       from dictionary.tables
       where libname="MYDIR" and modate>"&SYSDATE"d-1;
quit;
```

Note that this code uses the automatic macro variable SYSDATE and the date that the dataset was last modified. Automatic macro variables will be discussed later in the paper.

CREATE A FILENAME BASED ON CREATE DATE WITH (DICTIONARY.TABLES)

When receiving data files on a routine basis with consistent but not unique filenames, it is a good practice to change the name of the files to reflect the current version. For instance, if you routinely receive a file named mydata and the most recent file was created on August 1, 2005, it would be wise to change the name to mydata_01AUG2005 before you start using the data in order to differentiate it from the mydata file that was created on July 23, 2005 or July 16, 2005.

The code below uses the creation date field in DICTIONARY.TABLES to rename a file in the import directory called myfile. The filename is changed to myfile followed by an underscore and the file's creation date.

```
proc sql noprint;
       select crdate
       into :newcrdate
       from dictionary.tables
       where libname="IMPORT" and memname=”MYFILE”;
quit;

*** reformat the macro variable;
%let newcrdate=%cmpres(%substr(&newcrdate,1,7));

*** change the dataset name;
proc datasets library=import;
       change myfile = myfile_&newcrdate;
run;
```

If myfile was created on June 30, 2005, it would be renamed myfile_30JUN05. The PROC SQL step assigns the creation date (CRDATE) to the macro variable NEWCRDATE. The %LET statement reformats the value, dropping trailing spaces and time details. Finally the PROC DATASETS step changes the name of the file with the CHANGE statement.

FINDING MISSING VALUES IN A DATASET (WITH DICTIONARY.COLUMNS)

After determining what variables a dataset contains and how many observations are expected, one of the first questions asked is how many values are missing for each of the variables. Without SAS DICTIONARIES and PROC SQL, finding the counts of missing values for a list of character and numeric variables would require formats and frequencies and the output would be displayed across a number of pages. With PROC SQL and SAS DICTIONARIES, a very elegant solution that displays in a compact format is available. The solution below was posted to SAS-L October 2, 2003 by Ya Huang.

```
proc sql noprint;
       select 'nmiss('||compress(name)||') as '||compress(name)||'_Missing'
        into :mislst separated by ','
       from dictionary.columns
       where libname='WORK' and memname="MYFILE"; *must be capitalized;

       create table mistbl as
       select count(*) as TotalRecords, &mislst
       from myfile;
quit;

*** display output;
proc print data=mistbl noobs;
       title "Fill Rates for Dataset 'myfile'";
run;
```

The first SELECT statement in the PROC SQL step queries the DICTIONARY.COLUMNS table to find the variable names in the SAS dataset named myfile. Note that the values in the LIBNAME and MEMNAME fields are stored in all capital letters. These names are then used to construct pieces of SELECT statements which are saved to the macro variable MISLST for use in the second SELECT statement. The variable x would create the following value in the macro variable list:

```
       nmiss(x) as x_Missing
```

The second select statement uses these code snippets to query the dataset MYFILE. Finally, the results are displayed with a print statement. When applied to a table with two variables x(a numeric variable) and y(a character variable) each with three missing records, the following output is produced.

TotalRecords	x_Missing	y_Missing
13	3	3

This solution is much simpler than dividing the problem into numeric and character pieces. Also, The final output is very clean.

DICTIONARY.COLUMNS contains other information about variables as well, including datatype, length, position, label, and format. This is a particularly useful table as it houses just about anything that a programmer would need to know about the variables in a SAS dataset.

DYNAMICALLY FIX FORMATS (WITH DICTIONARY.COLUMNS)

Another common problem in processing large quantities of someone else's data is revising their formats to fit your needs. For example date variables frequently need date formats applied. This can be a non-trivial task when date variables are identified by text in their associated labels and you must manually review hundreds or thousands of variables. The DICTIONARY.COLUMNS table allows you to perform such a review automatically.

Suppose that you receive a dataset with a number of date fields. The date fields are identified in the labels with the word "date" but the date variables cannot be easily identified by their names and they lack any date formatting. The code below identifies variables in the dataset MYFILE that contain the word "date" in the label.

```
       proc sql noprint;
              select name
               into :dvarlst separated by ','
              from dictionary.columns
              where libname='WORK' and memname="MYFILE"
               and upcase(label) contains "DATE";
       quit;
       proc datasets library=WORK;
              modify myfile;
              format &dvarlst mmddyy10.;
       run;
```

These date variables are identified by the searching their labels for the word date. The date variables identified by the query are saved into a comma separated list and stored in the macro variable DVARLST. That list is then used in the DATASETS step that follows where their formats are updated. As you can see, it would be easy to alter the code to look at formats or variable names to find date references. It would also be easy to loosen the restriction and look for variations on DATE via wildcards with the LIKE keyword.

OTHER DICTIONARIES

This section has only showed you to two of the SAS DICTIONARY tables. SAS has eight others available to programmers: CATALOGS, MACROS, OPTIONS, MEMBERS, TITLES, VIEWS, INDEXES, and EXTFILES. The information contained in these tables should be fairly intuitive, with a few exceptions: OPTIONS contains a list of all the SAS System options and their values, MEMBERS contains information on datasets, views and catalogs, and EXTFILES contains the values of all active filerefs in your SAS session. To quickly peruse the contents of any DICTIONARY, try looking at the corresponding datasets in SASHELP.

MASTERING CONTROL FLOW WITH RETURN CODES IN SAS AUTOMATIC MACRO VARIABLES

In addition to the system options found in the DICTIONARY.OPTIONS table, SAS provides many of automatic macro variables at runtime. As the name implies, automatic macro variables are macro variables created during your SAS session. They can be used like any macro variable that you might create. Included in the automatic macro variables are variables that contain return codes from previously executed code. Return codes are numeric values that provide information about what happened during the previously executed step. These codes can be used to control the flow of your program, changing what code is executed based on what happened in the prior step. Other automatic macro variables contain information about your system at runtime. Still others support SAS/AF development.

To see a complete list of your session's automatic variables, use the following statement:

```
   %put _automatic_;
```

This section will focus on some of the system automatic macro variables that can help you better control program flow. The examples below were developed using SAS 9.1.3 running on a Windows XP platform. AF and other types of automatic macro variables are outside the scope of the paper. Note that values will differ for *nix systems, but the differences should be relatively intuitive.

MASTERING CONTROL FLOW 1: TESTING FOR SUCCESSFUL FILEREF AND LIBREF ASSIGNMENTS

Prior to reading a file it is good practice to verify its existence. For code that you only once this is not a significant issue. However if you are automating your code to run regularly, it is a good idea to ensure that your libraries and files are where you expect them to be before you try to use them. SAS allows invalid librefs and filerefs to execute and either generate a note (default behavior) or an error (by setting the appropriate system option). There is a third option, where you can alter the flow of your program to accommodate failed LIBNAME or FILENAME statements or better still, take remedial action.

By default the LIBNAME statement adds a note to the log file indicating a successful or unsuccessful assignment. A LIBNAME statement where the reference exists such as:

```
       libname lexists "C:\Documents and Settings\";
```

adds the following note to the log:

```
       NOTE: Libref LEXISTS was successfully assigned as follows:
             Engine:        V9
             Physical Name: C:\Documents and Settings\
```

Referencing a directory that does not exist such as:

```
       libname labsent "C:\nothere\";
```

adds a note to the log similar to this:

```
       NOTE: Library LABSENT does not exist.
```

Beyond writing notes to the log, the LIBNAME statement assigns a return code to the automatic macro variable SYSLIBRC. SYSLIBRC is set to zero with a successfully assigned libref or a non-zero number when the libref is not assigned successfully. (Note that zeros are the conventional return code value for a successful operation.) When preceded by the successful LIBNAME statement above, the following line:

```
       %put LIBNAME Return Code: &syslibrc;
```

wrote this to my log:

```
       LIBNAME Return Code: 0
```

Conversely, an unsuccessful LIBNAME statement wrote this to my log:

```
       LIBNAME Return Code: -7000
```

The automatic macro variable SYSFILRC behaves similarly for file references.

It is easy to extend your code to take some remedial action when a libref is not successfully assigned. The macro below creates the directory "C:\necessary\" if it doesn't already exist.

```
       %macro libmake;
              *** create the library if it doesn't exist ***;
              libname lneeded "C:\necessary\";
              %if &syslibrc ne 0 %then %do;
                     x "mkdir c:\necessary";
                     libname lneeded "C:\necessary\";
              %end;
       %mend;

       %libmake;
```

You could just as easily halt processing or perform some other action.

MASTERING CONTROL FLOW PART 2: ENSURING THAT OS COMMANDS EXECUTED CORRECTLY

Suppose that you want to test for the proper execution of an operating system command executed through the X statement. When operating system commands are executed it is expected that the command executes properly. SAS does not automatically test to ensure that OS commands behave as expected. Failed OS commands do not cause SAS to generate errors. This can be readily fixed with the use of the SYSRC macro variable. This variable contains the return code from the previously executed OS command.

Suppose that you want to download data from an FTP site on the internet for later use by your SAS program. The code below uses the x statement to execute an FTP script called cdc.ftp, which anonymously logs in to the CDC FTP server and retrieves a vaccine database.

```
       options noxwait;
       data _null_;
              *** get data from an ftp site using the ftp command ***;
              x "ftp -A -s:cdc.ftp ftp.cdc.gov";
              if &SYSRC ne 0 then ERROR "System Command did not execute properly";
       run;
```

This code will assign a value of 0 to &SYSRC and therefore will not generate an error message to the log. Note that the noxwait option should precede the X statement in windows to ensure that DOS windows are not opened and waiting for responses.

Now suppose that you have an erroneous FTP script name, referring to it as cdc.txt rather than cdc.ftp. The following code has a mistake in the FTP command:

```
data null;
       *** ftp attempt w/ incorrect script name;
       x "ftp –A -s:cdc.txt ftp.cdc.gov";
       if &SYSRC ne 0 then ERROR "System Command did not execute properly";
run;
```

This DOS FTP command will generate an error because a script was referenced that couldn't be found. The code will set _ERROR_=1 and print the line "System Command did not execute properly" to the log.

One cautionary note when using SYSRC, like SAS unexpected behavior of other programs is not necessarily an error. For instance, suppose you are trying to FTP a site anonymously but neglect to include the required anonymous parameter "-A" in the command line.

```
data null;
       *** ftp attempt w/o anonymous login - DOESN'T GENERATE EXPECTED ERROR;
       x "ftp -s:cdc.ftp ftp.cdc.gov";
       if &SYSRC ne 0 then ERROR "System Command did not execute properly";
run;
```

You will not successfully log onto the remote host (and hence fail to download the database) but the return code will still be 0 indicating a successful completion of the script. This highlights a very important rule when working with operating system return codes: make certain that you understand how the program you are calling generates return codes. Documentation is a useful starting point, but be sure to thoroughly test a range of possible problems.

MASTERING CONTROL FLOW PART 3: WORKING WITH SAS CONDITION AND ERROR CODES

DATA and PROC steps save information about the success of their execution to the automatic macro variable SYSERR. As with most other return codes a value of zero indicates a successful execution and any other value indicates an unsuccessful execution.

Suppose that you attempt to merge two datasets where the BY variable had different lengths in each dataset. You might see something like this in the log:

```
22         data merged;
23           merge sample1 sample2;
24           by x;
25         run;

WARNING: Multiple lengths were specified for the BY variable x by input data
sets. This may cause unexpected results.
```

The variable SYSERR would have a non-zero value because something unexpected occurred in the step. The code below shows the return code for a merge where the BY variable had different lengths.

```
26         %put Step Return Code:  &SYSERR;
Step Return Code:  4
```

SYSERR is very useful if you are interested in conditionally executing steps based on the success or failure of earlier steps. Details for the various values of SYSERR can be found in the SAS documentation under SYSERR.

The automatic macro variable SYSCC contains the return code that would be sent to the operating system if SAS were to exit at that point. In the above example, SYSCC would be set to 4 when SYSERR was set to 4. Suppose that another step executed without error after the merge above. The return code assigned to SYSERR would be set to zero but SYSCC would still be 4. The value of SYSCC is translated to another value before being returned to the operating system. The specifics of the translations from SYSCC value to the return code sent to the operating system are not readily available. However, the interpretations for return codes sent to the operating system can be found in the SAS documentation for your operating system under "Return Codes and Completion Status."

Be aware that SYSCC is writable by the programmer. If you want SAS to ignore an error and send a successful return code to the OS, you could simply set SYSCC to 0. Note that you would also have to set options NOSYNTAXCHECK and OBS=MAX in order to resume normal processing. If you are interested in generating errors, it is best to use the ABORT statement rather than manipulating SYSCC directly. ABORT allows you to control the return code sent to the operating system in a very straightforward fashion.

MASTERING CONTROL FLOW PART 4: VALIDATING INPUT DATASETS

Another activity ripe for automation in the world of repeated data processing is verifying that the input data set that you receive on a regular basis has not changed formats and that it's record count has changed appropriately. Any programmer that has worked with the same data provider for an extended period has encountered situations where the data that you received is not necessarily the data that you expected. Fortunate programmers receive advance notice and a detailed description of the impending changes, and then there are the rest of us…

The SAS procedure PROC COMPARE is very useful for addressing this kind of problem, i.e. verifying that data are arriving with a particular structure. A lesser used aspect of PROC COMPARE is its manipulation of the automatic macro variable SYSINFO. PROC COMPARE sends a summary of its findings to an automatic macro variable called SYSINFO. Other PROCs and the DATA step can update SYSINFO as well, but this discussion will focus on interpreting the values that PROC COMPARE assigns to SYSINFO.

PROC COMPARE assigns an integer to SYSINFO after each execution. This number can be converted to a bit representation where each bit can be used to test for one of 16 different "findings" from PROC COMPARE. The code below converts &SYSINFO to a binary string for processing:

```
   *** perform comparison;
   proc compare …;
       …;
   run;

   *** assign return code to another variable for processing;
   %let comprc=&sysinfo;

   *** data step resets SYSINFO to zero;
   data _null_;
       comprc=&comprc;
       comprcbin=put(comprc,binary16.);
```

For example, suppose PROC COMPARE generates a return code of 4224. The bit representation would be "0001000010000000". Not the extra step of assigning SYSINFO to another macro variable called COMPRC. This was done because the DATA _NULL_ step resets SYSINFO to zero, thus preventing you from reading SYSINFO inside of the DATA step. The condition assigned to each bit is defined in Table 1 below:

Table 1. SYSINFO Return Codes Base SAS(R) 9.1.3 Procedures Guide

Bit	Condition	Description	Bit16. Representation
1	DSLABEL	Data set labels differ	0000000000000001
2	DSTYPE	Data set types differ	0000000000000010
3	INFORMAT	Variable has different informat	0000000000000100
4	FORMAT	Variable has different format	0000000000001000
5	LENGTH	Variable has different length	0000000000010000
6	LABEL	Variable has different label	0000000000100000
7	BASEOBS	Base data set has observation not in comparison	0000000001000000
8	COMPOBS	Comparison data set has observation not in base	0000000010000000
9	BASEBY	Base data set has BY group not in comparison	0000000100000000
10	COMPBY	Comparison data set has BY group not in base	0000001000000000
11	BASEVAR	Base data set has variable not in comparison	0000010000000000
12	COMPVAR	Comparison data set has variable not in base	0000100000000000
13	VALUE	A value comparison was unequal	0001000000000000
14	TYPE	Conflicting variable types	0010000000000000
15	BYVAR	BY variables do not match	0100000000000000
16	ERROR	Fatal error: comparison not done	1000000000000000

As you can see, the table provides information about quite a few interesting and useful comparisons. While details are obviously lacking (e.g. which values were unequal), the information provided is very well suited to determining whether some key expected conditions were met.

Returning to our example of processing data that is routinely received from a vendor, suppose that you want to ensure that the data have the same variables, with the same data types and lengths. Further suppose that you expect records to be appended. Using Table 1 above you can determine that the following bits are of interest: 5 (LENGTH), 8(COMPOBS), 11(BASEVAR), 12(COMPVAR), and 14(TYPE). To see if the lengths are equal, you look at Bit 5 which is in the 12th position. To see if the types are identical, you look at Bit 14 which is in the 3rd position.

The quickest way to determine the position for a particular Bit in Binary16 representation is to subtract the Bit number from 17.

Elaborating on the code above we can examine &SYSINFO to verify that our criteria are met.

```
*** assign return code to another variable for processing;
%let comprc=&sysinfo;

*** data step resets SYSINFO to zero;
data _null_;
    comprc=&comprc;
    comprcbin=put(comprc,binary16.);

    if substr(comprcbin,12,1) then put "WARNING: Variable lengths differ";
    if (1-substr(comprcbin,09,1)) then put "WARNING: No new obs are present";
    if substr(comprcbin,06,1) then put "WARNING: New dataset is missing a var";
    if substr(comprcbin,05,1) then put "WARNING: New dataset has new var";
    if substr(comprcbin,03,1) then put "WARNING: Conflicting variable types";
Run;
```

It is easy to see how this variable could free the programmer from manual intervention or checks prior to processing and stop processing at a very early stage in the readin program.

MASTERING CONTROL FLOW PART 5: USING SAS TO CONTROL OTHER PROGRAMS

Given the information available in the SAS system variables it is easy to see how SAS could be used as a scripting language for batch processing. For example, suppose that you want to execute two operating system commands and then one of two SAS programs depending upon the successful execution of the operating system commands. The actual steps for this process are listed below:

1. Create a directory to hold a file
2. Once the directory is created, move data from a server to the newly created directory
3. If the move occurred without error, execute a SAS program
4. Otherwise execute a different SAS program

Obviously, this could be done in a DOS batch file (.bat) or VB script, but why leave SAS if you don't have to? The code below can address the process flow described above without ever leaving SAS.

```
options noxwait;
x "mkdir temp";
x "copy data.txt temp";
%macro runit;
      %if &SYSRC EQ 0 %THEN x "sas process1.sas";
      %else %if &SYSRC NE 0 %THEN x "sas process2.sas";
      %else %put "PROGRAMMING ERROR";
%mend;
%runit;
```

If the copy command executed without error then process1.sas is executed, otherwise process2.sas is executed. It's important to remember that you aren't restricted to executing SAS statements here, you can execute any operating system command or invoke any other command line program.

ENHANCING SAS LOGS WITH AUTOMATIC MACRO VARIABLES AND OS INFORMATION

While SAS logs are already incredibly informative about what is happening during the compilation and execution of code, you are not limited to what SAS has chosen to put in the log files. In fact, with a handful of automatic macro variables and some simple interfaces to the operating system you can add incredibly useful information about the SAS program that you are running and its environment. While automated approaches are never sufficient for documenting code, they can ensure that very common careless mistakes like failures to update the filename, owner, or the last person to run the code do not occur.

OTHER INTERESTING INFORMATION IN AUTOMATIC MACRO VARIABLES

SAS provides a number of automatic macro variables that describe your environment. You can determine the operating system with SYSSCP which contains generic information about your environment (e.g. "WIN" for windows) or more specific information with SYSSCPL (e.g. "XP_PRO" for Windows XP Professional). You can determine the number of processors being used with SYSNCPU.

You can get the date that your program or interactive session started with SYSDATE and SYSDATE9 (which return dates in DATE7. and DATE9. formats respectively). SYSDAY provides the day of the week that the program or interactive session started. SYSTIME will provide the system time at the point at which it is called.

You can get SAS versions in just about any level of detail that you desire. SYSVER returns the two level version number (e.g. "9.1" for version 9.1.3). SYSVLONG4 returns the complete version information with a four digit year (e.g. "9.01.01M3P07282004"). SYSVLONG returns the same level of detail with a two digit year. This info is particularly useful when you want to employ the latest and greatest SAS functionality in your program but you still have to play nice with others less fortunate than yourself, using older versions of SAS. For instance, suppose that you want to use a hash rather than SQL to combine information in SAS. You could wrap the hash code in a macro to execute when the SAS environment supports it or default to a SQL solution otherwise.

You can determine whether SAS is running in interactive or batch mode with &SYSENV which returns 'FORE' and 'BACK' respectively. You can also determine the userid that the current session is executing under with SYSUSERID. This can be particularly useful for situations where users other than the author can run programs. You can have the log automatically capture the last person to run the program without relying on the user to update the program.

CAPTURING OS INFORMATION PART 1: USING &SYSPROCESSNAME TO GATHER INFORMATION FROM THE OS

With only a few hoops to jump through, SAS provides easy access to information about the SAS program itself at runtime. SAS's self referential ability is particularly useful in two cases: finding the current location of the SAS program being executed and improving the log with information about the program itself.

When running SAS in interactive mode, the environmental variable SYSPROCESSNAME takes the value of the process id (e.g. "4321") However when run in batch mode that same environmental variable becomes a text string that contains the name of the program (e.g. 'Program "C:\Documents and Settings\rpless\myprogram.sas"'.)

If you are working in an environment where people are collaborating on code but working in different locations, or where programmers are working have separate development and production servers, having your SAS program find it's own location can prevent unpleasant typographical errors from cropping up in your code.

The standard log file can provides you with information about how your code executes at runtime but no information about the file itself. As you saw above you can extract the name and location of a program running in batchmode with the SYSPROCESSNAME macro variable. That same information can be used as an argument to DOS command line functions to provide you with additional information about your file.

One particularly useful DOS command to get information about your program file is the DIR command. The DIR command can return the name of the volume, a list of directory contents with last written dates, and file sizes as well as a summary of the number objects and the total space used by the objects. A number of options can be used to modify the output including: show file owner (/Q), omit summary information (/B), show creation date (/TC), show date last accessed (/TA) as well as a few others. Some examples can be found in code in Appendix 1.

CAPTURING OS INFORMATION PART 2: SYSGET AND OS VARIABLES

One of the simplest ways to pull information from your operating systems is to call environmental variables from your operating system. These values can be set by the programmer at the command prompt or automatically by logon scripts. This can be particularly useful for IT conveying useful (but not necessarily critical information) to SAS users such as current available disk space, recent server performance metrics, etc. Suppose that the IT group uses the logon script to create an environmental variable called itmsg with the value "The H drive will be backed up on Monday 1AUG05 to media id HP20050801" The SYSGET function can retrieve the value of itmsg from the operating system and assign it to the variable SASITMSG like so:

```
   sasitmsg=sysget(itmsg);
   put sasitmsg;
```

Your log would display

```
   The H drive will be backed up on Monday 1AUG05 to media id HP20050801.
```

By default Windows also has a number of variables that could be useful including the location of various temporary file folders (TEMP and TMP), OS, NUMBER_OF_PROCESSORS, as well as others. To see all of the values simply type:

```
   set
```

at a DOS command prompt. Note that much of the system information is available through SAS automatic macro variables, but SYSGET allows you to access user and administrator defined variables as well.

PRELIMINARY STEPS AT SELF-DOCUMENTING CODE

Now that you've seen some of the information available to you via automatic macro variables and environmental variables, let's put them together and to develop some simple self-documenting code. Appendix 1. contains an approach to having SAS code run in batch mode describe itself. It can be placed in a directory with the program that you would like to document and referenced with an include statement like so:

```
   %include "./logplus.inc";
```

Remember that when you include SAS programs, by default the log file of the calling program does not contain the source code nor any messages related to the source code from included programs.

An example of what is actually written to the log file is shown below:

```
   *****************************************************************
   *      Filename: "C:\Documents and Settings\rpless\main_pgm.sas"
   *         Owner: rpless (OVATION)
   *       Created: 06/16/2005 at 10:47 PM
   * Last Modified: 06/16/2005 at 10:55 PM
   *        Run by: rpless
   *   SAS Version: 9.01.01M3P021605
   *****************************************************************
```

While the code is straightforward, a few items warrant elaboration. This program uses some automatic macro variables available in SAS but it really relies on the value of SYSPROCESSNAME in batch mode to retrieve additional information from the operating system. Basically, the included code is finding the name of the calling program via SYSPROCESSNAME, using that name to find out information about the SAS file that called it, and putting that information plus some other information from automatic macro variables in the log.

Once the macro variable MYFILE has been assigned the full path name of the file (in double quotes) with

```
   %let myfile=%substr(&SYSPROCESSNAME,9);
```

the statements that follow execute DOS commands and return the values to filerefs via the pipe command like so:

```
   filename barefn pipe "dir "&myfile" /b";
```

These pipes generate lengthy notes in log files, even when the statements are in included code. The nonotes option will prevent these notes from appearing in the log.

These FILENAME statements generate a fairly lengthy output to the fileref as well including less than useful header and footer information. After each of the filename statements in logplus.inc, there are DATA _NULL_ steps where that data are cleaned up and the information that I want to keep is saved to a macro variable. Finally, the %PUT statements write lines to the log file using the macro variables that were created by the program as well as some automatic macro variables described above.

As you can see this approach can save the programmer from making tedious updates to the SAS program while still capturing very useful information.

CONCLUSION

SAS provides an extensive number of objects to help the programmer code in the face of limited information about inputs as well as a group of macro variables that allow the programmer to control the flow of the program based on the execution of previous statements. With a bit of investigation, programmers can automate significant portions of their code, reducing effort and errors.

REFERENCES

Laffler, Kirk Paul (2004), *PROC SQL Beyond the Basics Using SAS*. Cary, NC: SAS Institute Inc.

SAS Institute Inc (2005), *SAS Language Dictionary, Version 9*. Cary, NC: SAS Institute Inc.

SAS Institute Inc (2005), *SAS Macro Reference, Version 9*. Cary, NC: SAS Institute Inc.

SAS Institute Inc (2005), *SAS SQL Procedure User's Guide, Version 9*. Cary, NC: SAS Institute Inc.

SAS-L

ACKNOWLEDGMENTS

I would like to thank my colleagues at the Ovation Research Group for their very thoughtful comments on early drafts of this paper.

CONTACT INFORMATION

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

Richard Pless
Ovation Research Group
600 Central Avenue

Highland Park, IL 60035
rpless@ovation.org
www.ovation.org

APPENDIX 1: LOGPLUS.INC

```
*-------------------------------------------------------;
* Author: Richard Pless
* Created: 16Jun05
* Purpose: Find file attributes and put into log
*-------------------------------------------------------;

*-----------------------------------------;
* gather system info via a pipe
*-----------------------------------------;
options nonotes;
* remove unnecessary word PROGRAM from SYSPROCESSNAME;
%let myfile=%substr(&SYSPROCESSNAME,9);
* I need the path in double quotes for DOS to find the file;
* the extra quotes are escape characters;

filename barefn pipe "dir "&myfile" /b";
data _null_;
    infile barefn;
    input details $80.;
    call symput(barename,details);
run;

*** find owner and group ***;
filename owner pipe "dir "&myfile" /Q";
data _null_;
    infile owner truncover;
    input details $80.;
    ownerpattern=prxparse("/(\w*)\\(\w*)/");
    datepattern =prxparse("/^(\d\d\/\d\d\/\d\d\d\d) *(\d\d:\d\d (A|P)M)/");
    datefound=prxmatch(datepattern,details);
    ownfound=prxmatch(ownerpattern,details);
    if prxmatch(datepattern,details) and prxmatch(ownerpattern,details) then do;
          call symputx('group',prxposn(ownerpattern,1,details));
          call symputx('owner',prxposn(ownerpattern,2,details));
    end;
run;

*** find create date ***;
filename ctime pipe "dir "&myfile" /t:c";
data _null_;
    infile ctime truncover;
    input details $80.;
    datepattern =prxparse("/^(\d\d\/\d\d\/\d\d\d\d) *(\d\d:\d\d (A|P)M)/");
    if prxmatch(datepattern,details) then do;
          * symputx is required to remove some unusual special characters that ;
          * generate a number of extra lines;
          call symputx('cdate',prxposn(datepattern,1,details));
          call symputx('ctime',prxposn(datepattern,2,details));
    end;
run;

*** find date last modified ***;
filename wtime pipe "dir "&myfile" /t:w";
data _null_;
    infile wtime truncover;
    input details $80.;
    datepattern=prxparse("/^(\d\d\/\d\d\/\d\d\d\d) *(\d\d:\d\d (A|P)M)/");
    if prxmatch(datepattern,details) then do;
          call symputx('wdate',prxposn(datepattern,1,details));
          call symputx('wtime',prxposn(datepattern,2,details));
    end;
```

```
run;

skip;
%put *****************************************************************;
%put *      Filename: &myfile;
%put *         Owner: &owner (&group);
%put *       Created: &cdate at &ctime;
%put * Last Modified: &wdate at &wtime;
%put *        Run by: &SYSUSERID;
%put *   SAS Version: &SYSVLONG;
%put *****************************************************************;
skip;skip;skip;
```