SCL for the Rest of Us: Nonvisual Uses of Screen Control Language

Michael Davis, Bassett Consulting Services, Inc. North Haven, Connecticut

Abstract

Because of the increasing popularity of the SAS® System to create Graphical User Interfaces (GUIs), many have come to believe that the language used with SAS/AF® and SAS/FSP®, called Screen Control Language (SCL), may only be used with visual objects. The truth is that SCL can be used in conjunction with or in place of the SAS macro language to automate SAS batch programs and to better interface them with the computing environment. Specifically, SCL programs may be submitted from within batch programs run on MVS and UNIX host computers. This includes even those host computers on which SAS/AF is not installed!

Introduction

This paper is an introduction to the subset of SCL that may be used for nonvisual programming applications. Topics covered include:

- when SCL is better than DATA steps, macros
- interfacing SCL entries with SAS code
- how to execute SCL in batch
- how to use the SCL online reference materials
- using SCL to get the properties of the OS
- controlling the execution of SCL code.

No previous knowledge of SAS/AF or SCL is presumed. Part of the paper is devoted to using the SCL debugger to learn how SCL programs work and debug those that do not work.

What Is SCL?

The acronym SCL is an abbreviation of Screen Control Language, which is a programming language available when SAS/AF (application facility) or SAS/FSP (full-screen product) is licensed. The identification of this programming language as Screen Control Language is unfortunate in that it implies that SCL must be used with an interactive visual component.

With the advent of FRAME catalog entries, it is possible, if not usually practical, to build SAS/AF applications without using SCL program code. Similarly, it is possible and very practical to build SCL programs that have no visual component. Such programs can be used in situations where it is not desirable or possible for a user to respond to screen items, such as in production processes run in batch.

Advantages and Disadvantages to Using SCL

SCL offers advantages when compared to its alternatives, such as the macro language. These advantages, which are elaborated on, include:

- fewer ampersands and no percent signs lead to less cryptic coding
- variable names may be up to 32 characters long
- availability of an excellent debugger
- rich set of functions, thoroughly interfaced with the operating system
- ability to use SCL lists to store values
- ability to hide executed program code from the application user.

However, there are at least two significant disadvantages to SCL when compared to the macro language. First, the macro language is available to every SAS user. To create an SCL program, SAS/AF must be licensed and installed on the computer on which development is to take place. While SAS Institute often includes SAS/AF in attractively priced bundles of SAS System products, SAS/AF is not universally licensed on all computers on which SAS programs are used.

However, the compiled SCL programs do not require SAS/AF to be executed. Thus, they may be ported and used on any computer on which the base SAS software has been installed. In addition to the obvious potential license cost savings, the cross-platform portability of SCL frees developers to create SAS programs on computers where interactive program development is supported and encouraged. The compiled programs can be ported and executed on computer systems where interactive SAS sessions are not possible or discouraged.

Second, unlike the macro language, SCL variables are resolved at compile time during a single pass of the SCL compiler. This is in contrast to macro variables, which are resolved recursively by the macro processor. The macro processor will make as many passes as may be required to resolve any symbols before the program is compiled and executed.

Because the timing of when symbols are resolved is deeply embedded within the SAS System, it is likely that the inability to resolve the names of SCL variables after compilation will persist. In situations where symbols must be resolved when a program is run, the use of macro variables and perhaps even macro language programs will be still be required.

The remainder of this paper explores these concepts further through seven structured examples.

Creating Your First SCL Program

Let us begin exploring the nonvisual uses of SCL by creating our first program, example one, EXMPL001.SCL. For this purpose, we need access to a computer on which SAS/AF has been installed. We also must be able to run the SAS System interactively through Display Manager.

This last requirement is because it is not currently possible to create an SCL catalog entry through batch execution.

Catalogs are a type of file created by the SAS System, in which one may store various types of information as catalog entries. To begin using SCL for nonvisual applications, we need only concern ourselves with one type of entry, the SCL entry.

This paper illustrates the creation and editing of SCL programs under Windows 95 using Release 6.12 of the SAS System. However, users should be able to replicate these activities on any computer and operating system that allows the user to run SAS/AF interactively. Nuances of the SAS System peculiar to a particular operating system are covered in the SAS Companion manuals. If online help is available, these nuances also appear in the Table of Contents section, also titled "SAS Companion for …".

It is a good idea to dedicate a separate directory and libref to the examples to be created as part of this paper. To do this, start by selecting the Libraries icon on the Program window toolbar. This icon appears in the upper-right portion of the SAS Display Manager screen. After selecting this icon, the Libraries window should appear. Selecting the New Library push button brings up the New Library window.

Assign the libref scldemo to the directory c:\scldemo. Because this directory did not exist previously, a confirmation window appears after selecting *Assign*. Select the Yes button to create the directory and close the window.

The next step is to open the Build window. From the pull-down menu, select *Globals*, then *Develop*, and then *Application builder*. The Build window will appear.

In the area labeled Libraries, select SCLDEMO. If there were any SAS catalogs in the SCLDEMO libref, they would appear after the SCLDEMO libref was selected. Create a new catalog, MYPGMS, by clicking on the right mouse button to bring up a menu. Select *File*, then *New*, and then *Catalog*. The Catalog window appears next.

Enter mypgms and select *OK*. At the bottom of the Build window, you will see "NOTE: Catalog SCLDEMO.MYPGMS created", which confirms that the MYPGMS catalog was created. Note that SAS catalogs under Windows (and OS/2) use the .SC2 file extension.

Continuing forward, create the SCL entry for our first program. After verifying that SCLDEMO library and MYPGMS catalog are still highlighted, move the mouse cursor and click on the right mouse button. Select *File*, then *New*, then *Entry*. The BUILD: New entry window appears.

In the text entry field labeled Entry name, type EXMPL001. Then click the left mouse button on the down arrow. Select the SCL entry type. Select *OK* to create the new entry. The Source window appears.

Enter in the following SCL program:

```
INIT:
   PUT "my first SCL program";
return;
```

The author colors section labels blue to improve readability. To change the color of program code text, select the code (highlight) as a block of text, and enter the COLOR MTEXT <*color*> command in the command bar.

Section labels are similar to labels used in SAS DATA steps and in other programming languages. When statements that can cause the SAS System to execute statements out of sequence are used, such as the GOTO statement, labels are used to point to the section of the program to be executed next. The boundary of a section is denoted with a RETURN statement.

It is good practice to place all SCL code within labeled sections. While some SCL statements may work outside of a labeled section, executable statements will not. This should not be surprising. How would the SAS System know when to execute a freestanding SCL statement?

When constructing SCL programs for visual applications, there are several statement labels whose use denotes when the SCL code within that section should be executed. However, this situation is greatly simplified when coding SCL for nonvisual applications. We can place all our code within the INIT section, which the SAS System runs once when the SCL entry is opened for execution. Additional labeled sections may be used when needed to clarify branching or when LINK or GOTO statements are used.

Before we can execute our new SCL program, we have to compile it. Compilation means that English-like or "high-level" program statements are translated into machine code or pseudocode, depending upon the computer language and environment being used. Ordinary SAS code is compiled immediately after it is submitted to the SAS System for execution (unless using the Stored Program Facility).

However, with SCL programs, we compile them once, and from then on, run the program without having to recompile them. This feature of SCL offers a second important advantage. By compiling from the Program Editor with the NOSOURSE option, we can distribute the compiled program only and keep the source SCL secret. For those who sell their SAS programs, this is a very important feature of SCL.

To compile an SCL program, type the command COMPILE or select the Compile icon. Move the mouse off the Compile icon, and you see the following message at the bottom of the screen:

```
NOTE:  Code generated for EXMPL001.SCL.
                Code size=332
```

This means that our program was compiled successfully. If it had failed to compile, or if there were warnings, we would consult the Message window to learn the details. Open the Message window by entering the MSG command or by selecting the Message icon (if available).

In the interests of good housekeeping, change the description of the EXMPL001.SCL catalog entry to something a bit more descriptive. While still in the Build window, select the EXMPL001.SCL entry with the mouse and then right-click to bring up a menu. Select *File* and then *Rename*. This will bring up the Rename window.

Select the entry description and type in the highlighted area a more meaningful description similar to the one shown. Then select the OK button to return to the Build window. Close the Build window. To run the program, type and submit the following command:

```
AF c=scldemo.mypgms.exmpl001.scl
```

This command invokes SAS/AF. The C= parameter further specifies that the catalog entry to be executed is EXMPL001.SCL in the MYPGMS catalog stored in the directory associated with the SCLDEMO libref. The output of this program appears in the SAS Log as

```
my first SCL program
```

Admittedly, this is not a very interesting program. However, we have to start somewhere on our way to learning to tap the more powerful features of SCL.

Learning About SCL Online

Your next question might be, "How do I learn more about the features of SCL?" The obvious answers might be to read a SAS Institute publication or take a course. Those are good methods but they often involve spending additional funds, waiting, or both! Consider another possibility-- using the online Help system.

Using the online Help system as a reference to the SCL language requires that you already understand DATA step programs and functions. The immediate need is to identify the types of functions that are available to SCL programmers and the arguments and syntax associated with them.

To get to the list of SCL functions organized by category, pull down the Help menu selection and choose *SAS System*. When you see the Contents listing, the temptation would be to choose Application Development with SAS/AF Software or Screen Control Language. Resist that temptation! Instead, select *SAS System: Main Menu*. Select *Limited Index*, then *SAS/AF*, then *SCL*, then *Syntax*. Finally, select *SCL Elements* to bring up a list of SCL functions organized by categories. Please note that the help menus differ on some platforms.

SCL elements are functions available to SCL programmers. Some of these functions are not available within the DATA step. Note that you can also use most DATA step functions within SCL programs. Those functions are listed in the online help for SCL syntax under SAS Functions.

As nonvisual SCL programmers, we can safely ignore the categories of functions that relate to screen elements and the more esoteric features of the SCL language. These include:

- Choice Group
- Cursor
- Display
- Extended Tables
- Field
- Graphic Options
- Keys
- Legend
- Method Block
- Misc
- SAS/FSP
- Object Oriented Programming
- Window

The **Character** functions supplement those inherited from the SAS DATA step language. The only CONTROL statement needed for nonvisual SCL is CONTROL ASIS, described later. **Data File** and **Variable** functions provide a direct interface to SAS data sets and data set variables. **Directory** functions provide the names and other attributes for the members of directory files. **External File** functions allow the programmer to create, append, edit, and delete external (flat) files. **Formatting** functions transform SCL variables using SAS or user-created formats.

One of the most powerful features of the SCL programming language is the SCL list, introduced with Release 6.07 of the SAS System. An SCL list is a collection of information items, stored in memory. The number of items can be expanded at will. The items contained in an SCL list need not be of the same type. Character, numeric, and sublists may be stored in SCL lists. The items in a list may be retrieved by their position or name. List items may be searched, sorted, rotated, reversed, and popped.

What does one do with an SCL list? Lists are a convenient means for storing and manipulating the values obtained from interrogating SAS data sets and directories. The contents of SCL lists can be stored to and retrieved from SLIST catalog entries. Thus, they become an ideal way to pass parameters from one SAS session to another or from one computer platform to another.

Similar to macro variables, an SCL list can be local or global. When an SCL list is placed in the global environment, the items created during the execution of an SCL catalog entry are available to other SCL entries run during the same SAS session.

Because SCL programs are compiled in a single pass before they are run, they cannot write themselves as is possible with the macro language. Hence, **Macro** functions provide an important interface to macro variables. Similarly, the **Message** functions provide an interface to SAS System messages and return codes that may be generated during program execution.

Considering interfaces further, while most of the DATA step language can be emulated using SCL, SAS procedures often cannot be emulated. DATA steps and procedures may be sent to the SAS Supervisor for execution by placing the statements to be run within a SUBMIT block. SCL or macro variables may be referenced from within a SUBMIT block by prefixing the variable to be substituted with an ampersand (&).

To learn more details about any SCL function, from the Help Table of Contents, select *Screen Control Language*, then *SCL Dictionary*. A detailed description is shown when a function is selected.

A Second Example: Does a Data Set Exist? How Many Observations?

SCL is often used to test to see if a SAS data set exists. If the data set does exist, SCL may be used to gather information about it to control a larger program. Consider Example 2.

```
/*                EXMPL002.SCL              */
/*      Test data set for existence and     */
/*             # of observations.           */

INIT:

/* Test to see if SCLDEMO.BASEBALL exists. */
rc=libname('SCLDEMO','C:\SCLDEMO');
baseball_exist=exist('SCLDEMO.BASEBALL');

/* If SCLDEMO.BASEBALL exists, get the      */
/*       number of observations.            */
if baseball_exist then do;
   ds_id=open('SCLDEMO.BASEBALL','IS');
   nobs=attrn(ds_id,'NOBS');
   put 'Number of observations in ' ||
       'SCLDEMO.BASEBALL= 'nobs;
   rc=close(ds_id);
end;

/* Next line suppresses compiler warning    */
/*     that rc is not used.                 */
rc=rc;
return;
```

In this example, we first test to see if a sample SAS data set containing 1986 baseball player statistics is contained in the directory allocated to the SCLDEMO. For the purpose of illustration, the SCLDEMO libref is allocated using the LIBNAME function even though it was already allocated by the first example. The EXIST function returns a 1 if the SCLDEMO.BASEBALL data set exists.

Notice that this program takes advantage of the 32-character limit for the names of SCL variables. The variable used for the return code, BASEBALL_EXIST, uses this feature to improve the readability of this program. If the SCL variable is used to reference a SAS data set variable, the SCL variable name should be kept to eight characters for compatibility purposes.

Because the SCLDEMO.BASEBALL data set exists, the next section of code is executed. The SCLDEMO.BASEBALL data set is opened using the OPEN function. Because we don't need to change the data set or access any observations directly, we use the IS option to open SCLDEMO.BASEBALL in input-only, sequential mode.

The data set id, DS_ID, is a numeric variable set by the OPEN function. It is used to reference the SCLDEMO.BASEBALL for the ATTRN function. We use the ATTRN function to obtain numeric attributes, such as the number of observations (NOBS) in a data set. The message placed in the SAS Log appears as

```
Number of observations in
SCLDEMO.BASEBALL=  322
```

After putting the number of observations out to the SAS Log, we close SCLDEMO.BASEBALL using the CLOSE function.

Before considering our next example, we might comment on the question, "Why not use %SYSFUNC to accomplish this through a DATA step?". In Release 6.12, we could have used the %SYSFUNC macro language function to accomplish these tasks. However, the advantages of SCL, such as long variable names, easier to read code, and the SCL debugger would be lost.

Third Example: Listing Valid Subdirectories

SCL can be a helpful tool to get information about the environment under which the SAS System is being run. Because it is interfaced with that environment while the computer is operating, it is possible to accomplish tasks that cannot be easily accomplished by other means.

For example, when using SAS on a directory-based platform such as Windows, you might want to get a list of available subdirectories. Many SAS programmers faced with this problem might pipe the output from the DIR command to a file and then use SAS to parse the file for directory names.

At best, it is a nuisance to make this approach work. If the directory listing shifts or changes for any reason, the program will fail, perhaps without any warning! Further, it is not always clear what is a directory and what is a file. You could check for the presence or absence of a file extension. However, some files lack file extensions, while some subdirectories have them. Last, if the application is moved to another platform (for example, from Windows to UNIX), the program will probably require changes.

The SCL program shown in Example 3 overcomes these problems. Notice that it is built entirely from SCL functions. There are no X command calls to the operating system (for example, the DIR command). The program first opens the root directory for reading. If the directory open fails, the program puts a warning message to the SAS Log and then stops.

Next, the SCL program collects a list of all directory members into the SCL list TMP_LST. To illustrate the operation of this program better, the SCL list is written to the SAS Log for inspection. Then the program attempts to open each directory. Those that were successfully opened are placed in the SCL list DIR_LST. DIR_LST is then written to the SAS Log. Last, DIR_LST is written to an SLIST catalog entry so it can be read and used by other SCL programs.

Note that the absence of percent signs (%) and ampersands (&) makes this program easier to read and comprehend. Further, when run in the SCL debugger (covered later), the program logic can be traced and the SCL variable values may be inspected through the PUT and PUTLIST debugger commands.

One coding nuance is worth a comment. In the preceding SCL program, we determine if a directory entry is valid by inspecting the returned value of TST_ID, which is the test directory identifier (a number). Any directory identifiers not equal to zero tell us that the directory member is not a directory.

However, were we to use this technique to test the results of an operation where a write might be performed, a return code not equal to zero might be acceptable. For example, in the case of libref assignments, the return code -70004 tells us that the directory is already assigned to another libref but is available for reuse.

It would be a fair comment that a list of directory members (subdirectories) is unlikely to be used outside of a visual application using PROGRAM or FRAME entries. However, the technique demonstrated in this example could be adapted for use within an unsupervised SAS program submitted in batch in order to trap situations where a directory is unexpectedly made unavailable.

An example of this situation would be where a network fails and a drive assignment mapped to a file server is temporarily invalid. By testing before the next SAS step fails, the batch program can be cleanly terminated, saving computer cycles and avoiding potential data set corruption or over-writing. If a more sophisticated approach is desired, our program could pass an e-mail or console message to alert an operator of the problem. We can even save the current program status to a SAS data set or SLIST entry to support an automated restart routine.

```
/*            EXMPL003.SCL                */
/*   Send the root subdirectory list      */
/*         to an SLIST entry.             */

INIT:

/* Make temporary directory nodes list.  */
tmp_lst=makelist();

/* Open root directory to make list.     */
rc=filename('CURNODE','C:\');
dir_id=dopen('CURNODE');
if dir_id lt 1 then do;
   put 'Directory Open Failed';
   msg=sysmsg();
   put msg;
   return;
end;

/* Get directory entries.                */
mbr_cnt=dnum(dir_id);
do i=1 to mbr_cnt;
   mbr_nm=dread(dir_id,i);
   rc=insertc(tmp_lst,mbr_nm,-1);
end;
/* De-assign fileref.                    */
rc=filename('CURNODE','');
call putlist(tmp_lst,
   'List of all Root Directory Members',1);

/* Test for valid subdirectories.        */
dir_lst=makelist();
mbr_cnt=listlen(tmp_lst);
do i=1 to mbr_cnt;
   dir_nm=getitemc(tmp_lst,i);
   tst_pth='C:\'||dir_nm;
   rc=filename('TSTNODE',tst_pth);
   tst_id=dopen('TSTNODE');
   if tst_id then do;
      rc=insertc(dir_lst,dir_nm,-1);
      rc=dclose(tst_id);
   end;
   rc=filename('TSTNODE','');
end;
rc=dclose(dir_id);
rc=dellist(tmp_lst);
call putlist(dir_lst,
   'List of Valid Subdirectories',1);

/* Save directory list to an SLIST entry.*/
/* Dummy attributes list id.             */
dummylst=0;
rc=savelist('catalog',
            'SCLDEMO.MYPGMS.EXMPL003.SLIST',
            dir_lst,
             dummylst,
           'List of Root Subdirectories');
rc=dellist(dir_lst);
rc=rc;
return;
```

Interfacing SCL Variables with SAS Code

When SCL is used in applications with a visual component, a common approach is to gather program parameters from the screen. However, when a nonvisual approach is desired, we can substitute a control SAS data set created by hand or through some form of SAS processing. The values in the control data set can be converted to SCL variables and passed to ordinary SAS programs. These programs may be contained within SUBMIT blocks. SUBMIT blocks are pieces of SAS program code placed between SUBMIT and ENDSUBMIT statements. SUBMIT blocks are passed without interpretation to the SAS Supervisor.

That SUBMIT blocks are not interpreted by the part of the SAS System that processes compiled SCL programs is an important feature to note. First the SAS code within a SUBMIT block is not checked for syntax. If there are errors, the SCL compiler will ignore them and they will not make their existence known until the SCL code is run. When you create SCL variables solely for the purpose of putting them into SUBMIT blocks, the following message may appear:

```
  WARNING: [Line xx]  Variable xxxxxxx is
           defined but not used
```

This message may be ignored or suppressed as we often do with the system return code variable (SYSRC or RC) by including a statement that sets the variable equal to itself (for example, RC=RC). Another feature of the SAS code within SUBMIT blocks is that if we run an SCL entry in the SCL debugger, we do not see step-by-step execution of the code within the block. However, we could copy any DATA step code within the SUBMIT block to the Windows clipboard, paste it in the Program Editor, and test it with the DATA step debugger.

How are the values of SCL variables passed to SUBMIT blocks? Within the SUBMIT block, prefix the name of the SCL variable with an ampersand (&). This raises two questions: is that not how we tell the SAS System that a variable is a macro variable to be substituted by the macro compiler? Also, how does the SAS System know what type of variable is represented by a variable prefixed by an ampersand (SCL or macro variable)?

The answer to both questions is that when an SCL entry with a SUBMIT block is processed by the SCL compiler, it looks to see if the variable prefixed by the ampersand is an SCL variable. If it is, the SCL compiler inserts a reference to the SCL variable in the SUBMIT block. Otherwise, the SCL compiler ignores the variable. At run time, the macro compiler attempts to resolve the variable prefixed by the ampersand.

You might ask, "Why bother to use SCL to submit ordinary SAS programs? I can text edit my SAS code to filter the input SAS data set to contain the desired values with an IF or WHERE statement. If I wish to run a SAS procedure for each distinct value of a variable in my SAS data set, I can employ a BY statement."

You could use text editing to control a program for each run. However, text editing SAS code before each run is a nuisance, and it opens up the possibility for all types of errors. Further, if your goal is to move the responsibility for running a program to an operations department or a junior staff person, having to edit the SAS code before submission makes that program a less attractive candidate to enter production.
Using a BY statement to obtain separate analyses for each value of the SAS variable is also a good approach. However, what if you need to filter the observations passed to a SAS procedure on an ad hoc basis? The following example illustrates this point:

```
/*           EXMPL004.SCL           */
/*   List statistics for pair of   */
/*       baseball teams.            */

INIT:

/* Must use variables prior to     */
/*   FETCH in order to work.       */
team1='';
team2='';

/* Read selected teams             */
dsid=open('SCLDEMO.SLCTTEAM','IS');
call set(dsid);
rc=fetch(dsid);
rc=close(dsid);

/* Make sure that indentation is   */
/*    preserved with SUBMIT block. */
control asis;

/* Print only the observations     */
/*  for the selected teams.        */
submit continue;
proc print data=scldemo.baseball;
   where team in('&team1','&team2');
run;
endsubmit;
rc=rc;
return;
```

In Example 4, we wish to print the observations from the SAS data set used in our second example, containing 1986 baseball player statistics for those players on either the Los Angeles Dodgers or St. Louis Cardinals. We want to show both teams within the same list, so the BY statement approach will not satisfy our requirement.

Our program first creates the SCL variables TEAM1 and TEAM2 by assigning null strings to them. Otherwise, the FETCH function a few lines later will not work unless we employ GETVARC functions. We open the control data set, SCLDEMO.SLCTTEAM with the OPEN function. The CALL SET statement links SAS data set variables to SCL variables with the same name and type. The FETCH function copies the values in the SAS data set variables TEAM1 and TEAM2 to the SCL variables by the same name. Last, our program closes SCLDEMO.SLCTTEAM after the FETCH function is executed.

The SUBMIT block contains a PROC PRINT statement, followed by a WHERE statement. Note that in the WHERE statement, single quotes have been placed around the SCL variables TEAM1 and TEAM2, which

have been prefixed by ampersands. If &TEAM1 and &TEAM2 represented macro variables, double quotes would have been required.
CONTROL statements are often used when SCL programs are used with screen displays and are largely irrelevant for nonvisual SCL applications. However, our example uses a CONTROL statement with the ASIS parameter. This tells the SAS System to preserve the indenting of the SAS program code typed within the SUBMIT block when it is submitted for execution and echoed to the SAS Log.

This SCL program is run by submitting the Display Manager command:

```
AF c=scldemo.mypgms.exmpl004.scl
```

This yields the following listing in the SAS Log:

```
1 proc print data=scldemo.baseball;
2 where team in
('LosAngeles','StLouis');
3 run;
```

Only the two teams that were selected appear in the PRINT procedure listing.

While the preceding example is contrived, the same logic can be adapted to situations where you might wish to print a monthly report only for accounts that had some activity during the preceding month. The control data set could be created by the SORT procedure, using the NODUPKEY option and the IN= option. The PRINT procedure (or other analysis procedure or DATA step code) could be placed in a loop that is executed once for each observation in the control data set.

This approach enjoys two potential advantages. First, SCL programs are often easier to comprehend and debug than macro language programs that perform similar functions. Second, when compared to the MERGE statement or SQL, SCL programs may yield greater computer efficiency. If the account history SAS data set(s) are indexed, the SAS Supervisor quickly retrieves the desired observations when a WHERE clause is used. With a MERGE or JOIN, intermediate data sets or tables may need to be constructed, which probably consume more computer resources.

Table Look-up Using SCL Lists and Named Items

When table look-up is required in an ordinary SAS program, and running short of memory is not a concern, the FORMAT procedure is often a good way to perform the look-up task. However, when coding in SCL and an ad hoc look-up table must be created, the FORMAT procedure may be less attractive, especially if it must be created via a SUBMIT block.

Fortunately, SCL provides a capable substitute, SCL lists with named items. Unlike the FORMAT procedure, we can create and use SCL lists where the look-up key, which is the item name, may be duplicated. However, Example 5, there is no duplication of keys.

In this example, SAS is used under MVS TSO. We wish to print a file to a remote printer and set the destination automatically based upon the userid.   Ordinarily, we would maintain the look-up table as a SAS data set, as in the previous example. However in this example, we create the table directly within the SCL program.

```
/*  EXMPL005.SCL - Print to a remote */
/*    printer based upon user id.    */

INIT:
/* Create the look-up SCL list.      */
prt_id_lst=makelist();
rc=insertc(prt_id_lst,'PT0001',-1,
   'USER101');
rc=insertc(prt_id_lst,'PT0005',-1,
   'USER102');
rc=insertc(prt_id_lst,'PT0006',-1,
   'USER103');

/* Select printer.                   */
user_id=symget('SYSUID');
list_position=nameditem(prt_id_lst,
    user_id);
if list_position then slctd_prt=
    getitemc(prt_id_lst,list_position);
else do;
   put 'Unknown User Selected';
   return;
end;

/* Create the PRINTOFF command.      */
x_cmd='PRINTOFF (MYPRINT) CLASS(A)
   DEST('||slctd_prt||')';

/* Issue PRINTOFF command.           */
rc=optsetn('XWAIT',0);
rc=system(x_cmd);

/* Echo PRINTOFF command to SAS Log. */
put x_cmd=;
slctd_prt=slctd_prt;
rc=rc;
return;
```

Assume that USER002 is the TSO user who has logged in. First, the SCL list, PRT_ID_LST, is created. Each list item contains the remote printer assignment for a user, which is the item's name. We get the TSO userid, USER002, by using the SYMGET function with the SYSUID argument. &SYSUID is the name of the automatic macro variable containing the TSO userid. SYMGET is used because we need the value for the userid at run time, not compile time.

To verify that USER102 has a remote printer assigned, the NAMEDITEM function is used. Because an item named USER102 exists, NAMEDITEM returns the position of the list item. In this example, the item is in the second position in the list so NAMEITEM returns the number 2. If the item named USER102 was not in PRT_ID_LIST, the message "Unknown User Selected" would have been put to the SAS Log, and the execution of the SCL program would have ended. In this example, the item's value, PT0005, is assigned to SCL variable SLCTD_PRT.

Next, the TSO PRINTOFF command is built as the SCL character variable X_CMD. The OPTSETN function is used to set the SAS system option XWAIT so that the native TSO window will close automatically as soon as the SYSTEM function has finished issuing the PRINTOFF command. So that the value of the external command passed to the SYSTEM function can be seen, it is echoed to the SAS Log using a PUT statement.

For reference, please note that SCL item names are stored in uppercase regardless of how they are entered. Also note that this example illustrates the use of SYMGET function as an interface to macro variable values. In applications where it is necessary to pass an SCL value as a macro variable at run time, use the SYMPUT function.

Using SCL When SAS/AF is Not Available

A frequent misconception about SCL programs is that SAS/AF must be installed on the computer where the program will be run. All you need is base SAS. Starting with Release 5.18, SAS Institute has supplied the DISPLAY procedure. Originally intended to support the Institute's CBT (Computer-based Training) offerings, PROC DISPLAY also provides run-time support for all types of SCL programs.

To illustrate the use of PROC DISPLAY, let us suppose that we wish to run our first example on a computer on which only base SAS is installed. We would submit the following program (assuming that the SCLDEMO libref has already been allocated):

```
/* EXMPL006.SCL - demo PROC DISPLAY. */

proc display =scldemo.mypgms.exmpl001.scl;
run;
```

This yields the same message that we saw earlier in the SAS Log:

```
my first SCL program
NOTE: The PROCEDURE DISPLAY used 0.27
seconds.
```

Please note that statements in SUBMIT blocks are not executed until the PROC DISPLAY has finished executing.

Using SCL in Batch Programs

When SAS users are asked why they have not made use of the SCL language, one frequent answer is, "I have to run my programs in batch mode." Certainly, the name Screen Control Language could easily cause people to believe that SAS must be run interactively to take advantage of its features. Actually, the reverse is true. SCL can supply the between run customization often required to put batch programs into production.

Before we consider an example that illustrates the preceding principle behind batch program automation, it may be useful to consider common reasons why programs are executed in batch mode. First, access to some computer resources, such as magnetic tape drives, must be scheduled. A program often requests such resources through the use of Job Control Language (JCL).

Second, some disk storage management systems archive data sets after a specified period of disuse to tape or other slow-access storage media. Because there is no way to know how long it will take to restore an archived data set, timesharing operating systems such as TSO will cause the file allocation to fail. However, if the allocation is requested during batch operation, the operating system can wait until the requested disk volume has been restored before scheduling the job for execution.

Last, batch-mode execution allows those who supervise large computer systems to shift computer requests from periods of peak usage through the use of job classes. Some classes allow users to submit batch jobs during the day for execution during the periods of low usage in the early evening or night. Other classes allow batch jobs to be executed according to the sequence in which they were submitted and their relative priority. These classes are set to allow only so many jobs with each class to be executing at any moment.

How can SCL help us deal with batch submission scheduling systems? First, SCL programs can create JCL statements and cause them to be passed to the batch job scheduling system. In most cases, tape and archived data sets must be allocated outside of the SAS System for our programs to run successfully. Equally important, SCL can supply the flexibility and intelligence required to permit unattended operation of our programs.

Consider our next example, in which an application is run on a mainframe that runs the MVS operating system. We need to allocate filerefs for any member of a partitioned data set (PDS) whose member names contain a specified character prefix. However, between the times that we request the program to be run, the number and names of the PDS members change. Prior to each run, we could inspect the PDS using ISPF and edit the program to allocate the filerefs through DD allocation statements.

A better approach is to write an SCL program that determines which PDS members meet our criteria and takes care of the fileref assignments for the SAS program that follows. Even better, the SCL program can automatically submit the entire program, JCL and SAS language statements together, for batch execution. The following example does this. For clarity, asterisks (****) mark those parameters that each user would need to adapt and they are shown in **bold**.

```
/*                 EXMPL007.SCL              */
/*    Batch submission from SCL template.    */

INIT:

/* Make temporary directory nodes list.      */
        tmp_lst=makelist();

/* Open directory to make list.              */
rc=filename('CURNODE','**** pds dsn ****');
dir_id=dopen('CURNODE');
if dir_id lt 1 then do;
   put 'Directory Open Failed on MVS Host';
   return;
end;

/* Put member names in list.                 */
mbr_cnt=dnum(dir_id);
do i=1 to mbr_cnt;
   mbr_nm=dread(dir_id,i);
   if substr(mbr_nm,1,4) eq
           '**** member prefix ****' then
           rc=insertc(tmp_lst,mbr_nm,-1);
end;
rc=dclose(dir_id);
rc=filename('CURNODE','');
call putlist(tmp_lst,
   'List of all MVS PDS Members',1);

/* Open file in which to write program.      */
rc=filename('INPGM',
   '**** temp program dsn ****',
   '',
   'DISP=(NEW,KEEP,DELETE) SPACE=(TRK,(2,1))
        LRECL=80 BLKSIZE= 4000 RECFM=FB');
file_id=fopen('INPGM','O');
rc=fput(file_id,
   "//**** JOB STATEMENT ****");
rc=fwrite(file_id);
rc=fput(file_id,
   "//STEP1 EXEC SAS,OPTIONS='ERROR=1'");
rc=fwrite(file_id);
```

```
/* Write DD statements.                         */
n=listlen(tmp_lst);
do i=1 to n;
   dd_name='RAW'||put(n,Z3.);
   mbr_nm=getitemc(tmp_lst,i);
   dsn='**** pds dsn name ****('||dd_name||')';
   dd_stmt='//'||dd_name||' DD DSN='||dsn||',
      DISP=SHR';
   rc=fput(file_id,dd_stmt);
   rc=fwrite(file_id);
end;
rc=fput(file_id,'//SYSIN DD *');
rc=fwrite(file_id);

/* Include SAS program.                         */
rc=fput(file_id,
   "%INCLUDE('**** SAS program dsn ****')");
rc=fwrite(file_id);
rc=fput(file_id,'/*');
rc=fwrite(file_id);
rc=fclose(file_id);
rc=filename('INPGM','');

/* Submit program to internal reader.           */
rc=filename('INPGM',
            '**** temp program dsn ****',
            '',
            'DISP=SHR');
rc=filename('OUTRDR',
            'A',
            '',
            'SYSOUT=A PGM=INTRDR RECFM=FB
               LRECL=80');
inpgm_id=fopen('INPGM','I');
inrdr_id=fopen('OUTRDR','O');

do while (fread(inpgm_id)=0);
   rc=fget(inpgm_id,buffer,80);
   rc=fput(inrdr_id,buffer);
   rc=fwrite(inrdr_id);
end;

rc=fclose(inpgm_id);
rc=fclose(inrdr_id);
rc=filename('INPGM','');
rc=filename('OUTRDR','');
rc=rc;
        return;
```

How does this program work? The logic used in the second example to identify the file directory members has been adapted to identify PDS members. That we can use the same SCL code when running either under Windows or MVS is a powerful illustration of the cross-platform portability of SCL. The output of this section is the creation of SCL list TMP_LST, in which we have placed the names of PDS members that meet the selection criteria.

Next, a new file, INPGM, is allocated to contain the program statements by using the FILENAME function. Note how the various allocation parameters are supplied as a character string, delimited by spaces. We open the record pointer to INPGM using the FOPEN function so that the FPUT and FWRITE functions may place program statements in it. The FPUT function places the requested character string in the output buffer, and the FWRITE function moves the buffer to the INPGM file.

The section labeled "Write DD statements" is a loop that writes as many DD allocation statements as there are member names in the TMP_LST SCL list. If desired, this loop could have been coded so that the members would have been concatenated into a single fileref. To make it easier to substitute different SAS programs after the SYSIN DD statement, the %INCLUDE statement is used. Remember that we can allocate additional files and librefs within the included program by using FILENAME and LIBNAME statements.

Then INPGM is closed and reallocated. This rewinds INPGM so that the record pointer is positioned to the beginning of file. The fileref OUTRDR points to the Internal Reader, which is the portion of the operating system that accepts various computer language statements for batch execution. The DO WHILE loop passes each statement in INPGM to the internal reader.

Next, close the record pointers to the INPGM and the OUTRDR file reference. Last, the INPGM and OUTRDR filerefs are freed. Note that in a production program, the INPGM probably would have been allocated using the UNIT= or MGMTCLAS= parameters so that it would be deleted after a specified interval.

Note that in this example, the batch program could have been written directly to OUTRDR. However, by first writing to a file, you can inspect the statements created by FPUT and FWRITE functions during development by opening the file into the Program Editor window.

Using the SCL Debugger

Releases 6.11 and 6.12 made a DATA step debugger available to SAS users. However, an SCL debugger has been supplied with SAS/AF from a much earlier time. When an SCL program does not work as intended but the programmer knows the sequence in which statements are executed, the use of PUT statements may be a faster means to debug the program. However, when the execution sequence is unknown or is suspect, the SCL debugger can be a godsend. Further, if your goal is to move on to visual SCL programs, the SCL debugger is an excellent way to learn when particular labeled sections are executed.

The SCL debugger is available when an SCL program has been compiled in the Build window while debug mode has been toggled on. When the SCL program does not contain any SUBMIT blocks (the exception is SUBMIT SQL), it may be run in the debugger by selecting the desired SCL entry and issuing the TESTAF command. SCL programs with SUBMIT blocks may be run in the debugger by exiting the Build window and issuing the AF command with the option DEBUG=YES.

The documentation for the debugger lists over 20 commands. However, only a few of these are commonly used. To step through an SCL program line by line, press the Enter key. To see the value of an SCL variable, type PUT or E or EX, followed by the variable name. To see the values in an SCL list, type PUTLIST, followed by the list name. To leave the debugger, type QUIT.

After an SCL entry has been thoroughly debugged, it should be recompiled with the debug mode toggled off. This will reduce the size of the SCL entry and help it to run faster. Also, experienced SCL programmers make debugging easier by coding compact SCL entries that invoke other entries using CALL DISPLAY or CALL GOTO statements. Only the entry that is suspect is compiled with the debug mode turned on, greatly speeding the debugging exercise.

Conclusion

SCL greatly extends the flexibility and scope of the SAS System. Ironically, many SAS programmers think SCL is too complicated for their use. Yet its similar appearance to SAS code should endear it to those who would sooner resort to the macro language, whose syntax and appearance is often more confusing. When we strip the SCL programming language to the essentials required for nonvisual applications, SCL often seems simple. With the encouragement of this paper, consider extending the range of your programming abilities by using SCL in both nonvisual and visual applications.

References

SAS Institute Inc. (1994), *SAS® Screen Control Language, Version 6, Second Edition*, Cary, NC: SAS Institute Inc.

SAS Institute Inc. (1997), *SAS® Macro Language, Reference, First Edition*, Cary, NC: SAS Institute Inc.

Notices

The paper was originally published in *Observations®: The Technical Journal for SAS Software* on 09Nov1997 and has been edited to meet the SUGI paper presentation guidelines. It has been reprinted with permission from SAS Institute Inc. A copy of this paper as originally published, along with the source code and data files can be viewed and downloaded at URL:

http://www.sas.com/service/doc/periodicals/obs/obswww05/index.html

The author wishes to thank Vicki Brocklebank, David Driggs, and Ann Rockett of SAS Institute's Technical Support Division, who helped debug the batch submission example. He also wishes to thank the following SAS Institute staff who carefully reviewed the completed article for technical and other errors: Leonid Batkhan, Linda Fineman, Dan Fowler, Kita Garrido, Gerry Nelson, Ann Rockett, Jim Ward, and Jon White. Last, the author wishes to thank Carol Davis of Bassett Consulting Services, who proofread the initial draft of this article, and his friend, Dan Gronell, Gronell Consulting Services, for some especially helpful comments.

Questions and comments should be directed to:

Michael Davis
Bassett Consulting Services, Inc.
10 Pleasant Drive
North Haven CT 06473-3712
Internet: Bassett.Consulting@worldnet.att.net
Telephone: (203) 562-0640
Facsimile: (203) 498-1414