Some Unusual SCL Techniques
Paulette Staum, Paul Waldron Consulting, Inc.

ABSTRACT

A variety of techniques can increase your effectiveness and efficiency when you use Screen Control Language (SCL). These techniques include:

1) using macros, program entries or method blocks as utilities
2) using templates for common types of programs
3) using SUBMIT blocks for convenience of coding or speed of execution
4) executing commands for a user
5) using SCL in batch jobs

Examples include:

1) a utility to count the number of observations currently available from a data set with a WHERE applied
2) creating a selection list based on a format
3) quicker input/output by using SUBMIT blocks
4) an automated procedure to add SCL programs to a catalog of FSEDIT screens
5) a utility to identify the library containing a format

INTRODUCTION

Screen Control Language (SCL) is a powerful tool for writing user-friendly interfaces. As a developer, you want to develop an interface that is user-friendly, consistent, and predictable. You want to develop the interface as efficiently as possible, and you want it to be easy to maintain.

This paper presents some techniques for programmers who use Screen Control Language. It assumes that you are somewhat familiar with SCL. For example, it assumes that you have basic knowledge of what a SUBMIT block is, what an extended table is, and what a method is. It focuses on the use of SCL in SAS/AF®, rather than SAS/FSP®, although many of the techniques are relevant to both environments.

The first two techniques are not unusual. They are really just good programming practice as applied in the world of SCL. However, they are crucial to my goal of helping you use SCL more effectively. The last three techniques are specific to SCL. Their effectiveness is surprising.

USING UTILITIES

The first technique is obvious. Whenever you find yourself writing similar code for the third time, you probably ought to be writing a utility. SCL makes it tempting to reproduce similar code in several places, because you can mark lines, store them, and then paste them into another entry easily and quickly. It’s easy to think you are solving your problems by copying code.

Unfortunately, sooner or later you will want to improve that code. (Or worse, you’ll find a bug in it.) If you have half a dozen versions of it scattered throughout your system, it will be time consuming to change them all. Resist the temptation to copy code. Take the extra time to develop a utility to perform the task. The utility might be an SCL macro, or a PROGRAM entry, or an SCL method. You’ll have to stop and think about the utility in general, about what parameters it should have, and how else the tool might be used. But you will save time in the long run, your system will be much more consistent, and it will be much easier to maintain.

How do you decide when you should use a macro, a method, or a program entry? Some considerations are:

1) If you need a display window for the user, you should be using a PROGRAM entry (or a macro which calls a PROGRAM entry).
2) If you need to submit code, you can not use a method.
3) If you only need to include a few lines of code, for example to check return codes, use a macro.
One example of a convenient utility is a macro called %scnobs. It counts the number of available observations in a data set. The obvious way to count observations is to use the SCL ATTRN function with the arguments NOBS and NLOBS. However, these functions only return the number of observations disregarding any WHERE clauses that have been applied. Frequently, you need the number of currently available observations after the WHERE clause has been applied. If the data set contains a numeric variable, a utility like %scnobs can use the VARSTAT function to count the currently available observations. (See Appendix A.)

**USING TEMPLATES**

The second technique is based on something you do informally all the time. Sometimes you write code which is very similar to code you wrote before, but a utility can not be flexible enough to meet all your needs. When you face this situation, you copy your own code and modify it.

Formalize that habit. Identify a type of program that you write often. Create a model of the type. Include every feature and error check that can enhance the model. Test it rigorously. Then call it a template and keep it with the rest of your tools in a catalog. If you develop any improvements, be sure to incorporate them in the templates.

Any type of screen is a candidate to become a tool or a template. As you add to your catalog of utilities and templates, you find yourself developing better applications faster. You have more time to try to solve the unique problems of an application, because you spend less time writing and maintaining multiple versions of the same code.

As an example, people often use PROGRAM entries as menus, because of their great adaptability. The menu can ask for passwords or parameters, assign libraries and open files. Each menu is unique, but there will be many characteristics which all menus should share. Develop and use a "menu template" for consistency and reliability. Copy it and use it as the basis for each menu PROGRAM entry. Your users get an improved interface, and you get an improved development process.

As another example, you might need to display a selection list, but either 1) there is no suitable SAS tool, or 2) you want more control than you have with the SAS selection list tools.

You might have a format with a list of one-to-one codes and decodes from which the user should choose. To display a format as a selection list, use PROC FORMAT with the CNTLOUT option to unload a copy of the format into a data set.

SUBMIT CONTINUE;
PROC FORMAT LIBRARY=frmtlib
    CNTLOUT=work.fmtlst
    (KEEP=START LABEL);
SELECT fmtname;
RUN;
ENDSUBMIT;

Then create a program entry with an extended table to display the data set created from the format as a selection list.

**USING SUBMIT BLOCKS**

SCL provides language elements that can perform data set and external file input/output functions which interactive systems often require. Data set lookups or appending observations are examples. Many of these functions are also available in base SAS. When are SUBMIT blocks better than SCL statements for data access?

Odd as it may seem, there are times when using SCL functions will be significantly slower than submitting a block of code. In general, SCL code is more efficient for processing smaller data sets or files. As data sets or files become larger, submitted DATA Steps or PROCs become dramatically more efficient.

For example, testing under IBM's MVS/TSO shows that SCL functions and submitted code take the same time to add 100 observations (of approximately 200 bytes each) to a data set. Similarly, SCL
functions and submitted code take the same time to add 30 records (of 80 bytes each) to a file.

As you would expect, SCL functions are faster for writing fewer records or observations. The time savings are quite small—approximately one tenth of a second.

However, when writing more records or observations, SUBMIT blocks become faster. The larger the output, the bigger the time savings offered by SUBMIT blocks. For the case of 1,000 small observations, tests showed a difference of 2.5 seconds. The relationship appears to be quadratic.

Development time is also a major issue, in addition to execution time. The coding to open and close a dataset or file is about the same for base SAS and SCL. However, base SAS generally requires less additional code for each observation or file written, as you can see in the example below.

Note also that any macro variables in SUBMIT blocks require two ampersands (&&) preceding them. A single ampersand refers to the value of a variable in the SCL program submitting the block of code.

There are significant first-time costs to using a DATA step in a SUBMIT block. Whenever a SAS/AF application is started, the first SUBMIT of a DATA step takes approximately a half second more than any subsequent SUBMIT block with a DATA step. Since starting a SAS/AF application normally only happens once per session, this cost is usually well worth accepting.

EXECUTING COMMANDS FOR A USER

CALL EXECCMDI provides the ability to execute a command immediately, as if it had been typed on the command line. CALL EXECCMD is comparable, but more temperate. It waits until control is returned to the user before executing the commands.

As a simple example, consider using these commands when you are displaying an extended table. Suppose the user wishes to move directly to a specific line of the table, and knows the value of the key variable on that line.

The user should type a KEYFIELD or NAME command to define the field to be searched. Then the user can type a FIND or LOCATE command to move to the line he wants.

However, you often know which variable should be searched. And requiring your users to know variable names is not user-friendly! You can name the variable to be searched, using:

```
CALL EXECCMDI('KEYFIELD varname;')
```

(Disadvantages are that CALL EXECCMDI does not work for this purpose, and this leaves a message on the message line.)

As another example, CALL EXECCMDI can also be used to create a pseudo-script. If you have a task which requires changing many entries in exactly the
same way, this technique can automate the procedure and save your sanity.

Suppose that you have a catalog of many FSEDIT screens which do not have SCL programs attached to them. You want a standard program attached to each screen, possibly to add a custom pmenu or to display customized help.

You do NOT want to call up each FSEDIT screen manually and modify it by including a template. How do you solve your problem? Create a PROGRAM entry to do the job.

First, the entry SUBMITs a PROC CATALOG to create a data set with the names of the entries in the catalog.

```
SUBMIT CONTINUE;
   PROC CATALOG CATALOG=&cat;
       CONTENTS OUT =entries;
   QUIT;
ENDSUBMIT;
```

Second, the entry uses a DO loop to fetch each observation of the data set. Each observation will represent one catalog entry. For each fetch, use a CALL EXECCMDI('...') to perform the required actions on the entry.

In this example, the commands to be executed are:

```
CALL EXECCMDI('FSEDIT dataset screen;
   MODIFY ; 3; INC template;
   END;END;END;').
```

(The 3 is needed to tell SAS that you want to modify the program for the screen.)

In this example, you need a template of the SCL program which each screen should have. You also need access to a data set with the correct variables for each screen. The example will not work if the FSEDIT screens already have a program with an FSETERM section.

This technique is limited to whatever you can do from the command line. However, SAS will resolve macro variables and macro calls typed on a command line, which adds even more flexibility to this approach.

**USING SCL IN BATCH**

There are some tools available in SAS/AF SCL which are not available in base SAS. It sounds incredible, but you can use SCL functions in batch jobs. You do it by using PROC DISPLAY to run SCL entries.

For example, create an SCL entry which gets information from the batch job via SYMGETs of global macro variables. The SCL entry uses the SCL language elements to perform the desired function. It then sets global macro variables via CALL SYMPUTs to return information to the batch job.

It would be nice to be able to use a local macro variable environment to communicate with an SAS/AF entry. However, SAS/AF entries only utilize the global macro variable table.

As an example, SCL functions can determine whether a specific format entry exists in a catalog. You can use the CEXIST function to determine which library's format catalog contains a format. (See Appendix B for code to use this feature in a batch environment.)

**CONCLUSION**

SCL can increase your efficiency and effectiveness as a developer dramatically. With a little experience and imagination, you can solve problems in ways that provide advantages to both the user and the developer.

I hope you find ways to apply some of these techniques to your own problems. I would be delighted to hear about some of your favorite techniques.
REFERENCES


2. Selby, Yvonne and Harris, Annette. 'Introduction to Extended Tables in SAS/AF Software Applications', SAS Observations, v 1, n 3 or 4, pp. 41-48.

ACKNOWLEDGMENTS

My thanks to Tom Hoffman of American Cyanamid, Dan Gronell of Pfizer, and Kenneth Chi of Hoffman LaRoche for their help in developing many of these ideas.

SAS, SAS/AF and SAS/FSP are registered trademarks or trademarks of SAS Institute Inc. in the USA and other countries. © indicates USA registration. ™ indicates trademark.

The author can be contacted at:
Paul Waldron Consulting Inc.
2 Tupper Lane
West Nyack, NY 10994

APPENDIX A - %scnobs

%macro scnobs(dsid,n);
/--------------------------------- 
Determine the number of observations in an opened data set.

Param s: dsid = data set id of dataset
 n = numeric variable to hold # of obs
    (will be set to . if error occurs)

Deleted observations will not be counted.
If the dsid refers to a dataset with a WHERE, then the number of observations will be the number satisfying the WHERE.

WARNING: Will not work if data set holds only character variables

_nobsname = "";
_nobsi = 1;
* find a numeric variable;
DO UNTIL (_nobsi > ATTRN(&dsid,'NVARS') OR 
_nobsname NE "");
   IF (VARTYPE(&dsid,_nobsi) = 'N') THEN
      _nobsname=VARNAME(&dsid,_nobsi);
      _nobsi + 1;
   END;
ELSE &n = .;  
END;

* find the sum of obs with and without values in var;
IF (_nobsname NE "") THEN DO;
   _nobsst=VARSTAT(&dsid,_nobsname,
                      'N', _nobsst);
   _nobsst=VARSTAT(&dsid,_nobsname,
                      'NMISS', _nobsst);
   IF (_nobsst > 0) THEN &n = .;
ELSE &n = _nobsst + _nobsst;
END;
ELSE &n = .;
%mend scnobs;
APPENDIX B - USING SCL IN BATCH

SCL entry:

```scl
/*--------------------------------------*/
NAME: FINDFMT.SCL
DESC: Find the first library in the list in &fmtsrch whose FORMATS catalog contains the format &format.
Communicates via macro variables:
 &fmtsrch &findfmt &fmtlib

Requires:
 &fmtsrch containing list of libraries with format catalogs
 &findfmt must be specified as format.entrytype, for example NAMEGC..FORMATC or ZIP.INFMTC
(can modify code to translate from PROC FORMAT style names)

Returns &fmtlib =
library logical name if format found
 (or) null if format not found
 (or) -1 if a library not found or other error

LENGTH format $17
 fmtsrch $200
 testlib $8

INIT:
 * retrieve format search path and format;
 fmtsrch = SYMGET('fmtsrch');
 format = SYMGET('findfmt');
 CALL SYMPUT('fmtlib','',1);
```

Batch Job:

```scl
i = 1;
DO UNTIL (testlib EQ ' ');
 testlib = SCAN(fmtsrch,i);
 IF (testlib NE '') THEN DO;
 IF (LIBREF(testlib)) THEN DO; "invalid lib;"
 CALL SYMPUT('fmtlib' ,-1);
 testlib = ''; "exit loop;"
 END; "invalid lib;"
 ELSE DO; "valid lib;"
 IF (CEXIST(testlib !.'FORMATS.' ! format)) THEN DO; "format found;"
 CALL SYMPUT('fmtlib',testlib);
 testlib = ''; "exit loop;"
 END; "format found;"
 END; "valid lib;"
 END; "scanned word exists;"
 i = i + 1;
END; "end of do until testlib blank;"
RETURN;
```

```scl
%global flldfmt fmtlib fmtsrch;
%let flldfmt=SEX.INFMT;
%let fmtsrch=PROJFMT DEPTFMT COMPFMT;
%let fmtlib=;

PROC DISPLAY
 C=PROGLIB.UTIL.FINDFMT.SCL; RUN;
%put Format &findfmt found in catalog &fmtlib.FORMATS;
```