No Program Is An Island: Passing Parameters to Tasks Launched by Servers and Schedulers

Michael L. Davis, Bassett Consulting Services, North Haven, Connecticut
Gregory S. Barnes Nelson, STATPROBE, inc., Raleigh/ Durham, North Carolina

ABSTRACT

Many computer tasks are controlled and modified by parameters. They convey to our tasks information such as what range of dates to include, which reports to run, where to find the raw data, and sometimes whether an error has occurred that should cause suspension of the requested task. Parameters are one of the main means for tying tasks together to form applications.

When using SAS/AF®[1] and SCL, parameters can be passed along with CALL statements, through SCL Lists and SLIST catalog entries, parameter data sets and files, and through macro variables. This paper will explore and contrast examples of these techniques. Strategies to choose the best means for passing parameters to different types of successor tasks will be offered.

When tasks are launched by a web server or a production scheduler, additional problems may emerge. How can we anticipate and accommodate changes in the computing environment between when a task is scheduled or requested, and when the task starts to run? Strategies to make server and scheduler launched tasks more robust will be shared.

INTRODUCTION

"No man is an island, entire of itself"

So wrote John Donne in 1623. So it is with SAS applications. Often the development of SAS® applications begins in the following manner. A need for an analysis is perceived and an analyst will go forth and cobble together a series of data steps and procedures that yields the desired listing.

Next, a similar need emerges. The previously run program is dusted off and edited to supply the current parameters. The revised program is again submitted.

At some point, the process breaks down. The analyst may tire of running the same program over and over again. The organization which commissioned the original program may wish to streamline the process so other analysts, or those with less computer training can obtained the program's output.

Another possibility is that programs developed in such a manner may become unwieldy. Good development techniques dictate that we dissect and disassemble such programs into modules, components, or objects (see Barnes Nelson, 1999). A good design increases the ability to reuse portions of the program and to more easily troubleshoot the process when things go awry.

When every element of the program is in the same module, the need to develop a scheme to pass parameters is lessened. Flags and arrays can be used to retain program states during execution. However, as one program concludes and a successor is started, techniques are often needed to tell the next program what to do. This paper is an exploration of the techniques available to SAS developers to meet this requirement.

Scope of this Paper

In this paper, we primarily focus on *execution* of programs and how we pass parameters to that program. In a larger context, it is helpful to understand where this fits in an application.

Initially, you need to be able to collect the information through the use of forms in an interactive application (e.g., SAS/AF FRAME or an HTML web form) or by populating a data set in which the parameter information is stored (and subsequently retrieved).

Next, you need to determine how long you want to retain the state of the parameter information. In other words, is the information going to be stored permanently or merely for the duration of the job?

Finally, you will need to decide on how the job gets executed. Should the program execute immediately, or scheduled for a later time (and possibly a separate machine / server?)

Although the scope of this paper focuses on the execution of jobs in which parameters are supplied, a note about the collection, storage and scheduling is briefly provided here.

DATA COLLECTION

The authors of this paper assume that you can "trap" the parameters you need in your programs. There are a variety of methods that might be appropriate depending on your needs. For interactive applications, you can easily use SAS/AF FRAME or PROGRAM entries to present the user with a form and selection criterion. For the brave of heart, you might even use %display statement to pop-up windows while executing macros. For programmers who are facile with other environments, you can use UNIX shell scripts that front-end the input; ISPF panels for MVS users; REXX programs on MVS and CMS; as well as a multitude of other methods.

---

[1] SAS, SAS/ACCESS, SAS/AF, and SAS/CONNECT are registered trademarks of SAS Institute Inc. SyncSort is a registered trademark of SyncSort Inc.

Despite the fact that these are interactive data collectors, this doesn't mean that you can't store this information permanently or even use this to schedule the job to run at a later time.

For non-interactive applications, you will need to use another method to capture the information. We will discuss many of these including write sequential files and even editing macro invocations (in the calling SAS program.)

HANDLING PERSISTENCE

Throughout this paper, we intimate that some programs will be run interactively, others will be run in batch, and in others, executed at some later date through a scheduling mechanism. Regardless of when we run these, we have to pay attention to the "state" of the application. Persistence is the result of remembering or tracking incremental changes in the state of an object, movement or action. That is, how long something stays "alive" and remembers where it has been. For example, let's say that we log into an application. The application will have to remember that the userid is `maddoggy`. Additionally, we may want to save some of our preferences from session to session.

We can use some of the techniques described in this paper to accomplish the task of persistence (e.g., store the userid in a global macro variable or an SCL environment list and retrieve it when needed.) However, some of the methods described here won't be good beyond the duration of the SAS session, so we have to store them permanently. Methods that we will discuss here might include storing the information in a data set, an SLIST entry or in a sequential file.

COUNTING THE WAYS TO PASS PARAMETERS

It should never come as a surprise to someone working with the SAS software that there is more than one way to accomplish a programming task. The authors have identified several techniques that they have used to pass parameters from one program module to successors. They include:

- global macro variables
- macro parameters
- CALL EXECUTE
- SAS data sets
- sequential text files
- ENTRY statement
- environment SCL lists
- SLIST catalog entries

Some of these parameter-passing techniques are applicable only when SAS/AF is being used. Also, the techniques are not mutually exclusive. Using multiple parameter passing techniques in concert can be extremely effective.

Next, we will explain how each method is used and describe each method's strengths and weaknesses.

Example Parameters Used

To make our code examples easier to follow, we will use a consistent example to illustrate the parameter passing technique being discussed. For our example, let us assume that we are creating an application to be used in a sales environment. We want pass the following parameters to a reporting application:

```
Title: 'GBN-MD Corp. Big Accounts'
Data Set:  'mydata.sales'
Accounts:  '00001', '00002', '00005'
Start Date:  '01JUL1998'd
Thru Date:  '31DEC1998'd
Notify:  'maddoggy@gbnmd.com'
```

In our example, Title, Data Set and Notify are character strings. Accounts represents a *series* of character strings. In this example, there are only three accounts but we want to account for any number of accounts that might need to be passed. Start Date and Thru Dates represent SAS date values, a special type of number in SAS. For clarity, we have represented Start Date and Thru Date as SAS date constants.

Global Macro Variables

The MACRO language within the SAS System is an extremely powerful and versatile tool. This fact is evident as we can use macro variables to pass information around in an application.

When using macro variables to pass parameters from one program to a successor, the macro variables referenced will have to be either global variables or system variables. It is important to note that local macro variables are not available to successor programs.

Macro variables may be set through %LET statement and the SYMPUT and SYMPUTN call routines. %LET statements can be used in open code. SYMPUT and SYMPUTN must be called from within a data step although that data step could be a DATA _NULL_ step. One pitfall to remember is that macro assignments created within a data step are not available for use until the step boundary, such as another DATA or procedure statement, is encountered.

A very effective use of macro variables is by front ending them through some collector program like a SAS/AF FRAME or PROGRAM entries. The collector application can then pass parameters to or export parameters to the program for execution. A common mistake is that if you forget to use the SYMGET function or SYMPUT call, your macro variable will contain the contents of the variable at compile time rather than at run-time.

What are some of the advantages of using global macro variables to pass parameters? First very straightforward coding is usually all that is required implement macro

parameters in a successor program. Second, macro variables in Version 6 can be as long as 32K while a data step variable is limited to 200 characters. It is not uncommon to pass as a parameter a macro variable longer than 200 characters so this is a very easy limit to bump up against.

To create the required macro variables for our example, we might submit statements in the first program such as:

```
/* make macro vars for example */
%let title='GBN-MD Corp. Big Accounts' ;
%let acct001='00001' ;
%let acct002='00002' ;
%let acct003='00005' ;
%let dataset=mydata.sales ;

data _null_ ;
 call symput('start', put('01JUL1998'd,9.)) ;
 call symput('thru', put('31DEC1998'd,9.)) ;
run ;
```

The data _null_ step is employed to turn our date constants into character strings of numbers representing the Start Date and Thru Date range (14061 through 14244). The following PRINT procedure shows how we would employ the parameters in a program:

```
/* print using parameters */
proc print data= &dataset ;
   title1 "&title" ;
   where account in(&acct001
         &acct002 &acct003) and
         saledate ge &start and
         saledate le &thru ;
run ;
```

The accounts passed in the WHERE statement deserve some comment. The IN function clause must provide for as many accounts as might be potentially encountered. If we had only two accounts, we would need to set the third macro variable, ACCT003, to blanks. The blanks would then be ignored when the IN clause is evaluated.

As our example illustrates, macro variables can be used to pass parameters to a second program while still in the same session. However, other techniques are required when macro variables have to be passed to program run in a second SAS session, such as a program executed in batch mode. Macro variables also tend to get awkward when the number of list items (array elements) varies from run to run as in our Accounts example.

Macro Parameters

In the context of parameter passing, macro parameters used in conjunction with a macro code function in a manner similar to global macro variables. The main difference is that it is not necessary to use %LET statements or SYMPUT calls to create macro variables passed as parameters. Our previous example might be implemented as follows:

```
data _null_ ;
  call symput('start', put('01JUL1998'd,9.)) ;
  call symput('thru',  put('31DEC1998'd,9.)) ;
run ;

%macro praccts(dataset,title,
    acct001,acct002,acct003) ;
   /* print using parameters */
   proc print data= &dataset ;
     title1 "&title" ;
     where account in(&acct001
           &acct002 &acct003) and
           saledate ge &start and
           saledate le &thru ;
  run ;
%mend ;

%praccts(sales,
        'GBN-MD Corp. Big Accounts',
        '00001', '00002', '00005')
```

One concept that might be noted from the preceding example is that global macro variables, such as the Start Date and Thru Date variables can co-exist with macro parameters.

When should macros be used with parameter passing? One situation is where the parameters need to be resolved or manipulated. Within a macro, additional functions are available for this purpose. Another situation might be where the parameters are used to select the portions of program code that must be run.

What are some of the advantages of using macro parameters as a parameter passing mechanism? Macros offer the ability to pass blocks of code to be executed as well character and number values. Macros can evaluate themselves. Consider the following example.

```
%macro dubldo(numvars) ;
data test ;
   %do n = 1 %to &numvars ;
      var&n = 1 ;
   %end ;
run ;
%mend ;
```

The macro call `%dubldo(3)` creates a data set named Test with the variables var1, var2, and var3.

By contrast, it would be awkward at best to write an SCL (Screen Control Language or SAS Component Language in Version 7) program that would replicate the ability to create dynamically create data set variables as shown in the preceding example.

One disadvantage to using macro parameters is that the resulting programs can be difficult to debug. Because of the need to provide for the ampersand when the macro parameter is tokenized, the length of macro parameters to names in Version 6 is restricted to only 7 characters.

The number of macro parameters usually needs to be laid out at the time the macro is coded. Last, as

previously observed, the handling of parameter lists (array elements) of varying lengths can be a problem.

Using CALL EXECUTE to Pass Parameters

Another portion of the macro facility is the EXECUTE call routine. The EXECUTE routine is called from within a data step. It passes the data step and passes the argument within the parenthesis to the macro facility for processing.

Any SAS statements generated while the CALL EXECUTE processes run after the current data step has completed. A nifty feature of the EXECUTE routine is that it will work regardless of the setting of the Macro system option.

Where does the EXECUTE call routine figure in the discussion of parameter passing? We can use this routine to incorporate step value values into macro calls. Consider the following example.

```
data accounts ;
  input @1 account $5. ;
datalines ;
00001
00002
00005
;

%macro praccts(acct) ;

/* print using parameters */
   proc print data= mydata.sales ;
     title1
      'GBN-MD Corp Big Accounts' ;
     where account eq "&acct" and
      saledate ge '01JUL1998'd and
      saledate le '31DEC1998'd ;
  run ;
%mend ;

data _null_ ;
  set accounts ;
  call execute('%praccts(' ||
    account || ')') ;
run ;
```

In this example, separate PRINT procedure listings will be produced for each of the three observations in the Accounts data set.

As the preceding example illustrates, the EXECUTE call routine is a good way to create programs that must execution conditionally or cycle through each observation in a control data set. It does not address the awkwardness that we commented on earlier where a list of parameters must be processed during a single iteration, such as an IN function clause.

Passing Parameters via SAS Data Sets

The preceding example, which illustrated passing parameters using the EXECUTE call routine also illustrated use of SAS data sets as mechanism for passing parameters. There are two primary variations of using SAS data sets as a parameter passing technique.

- each item in a list is placed in a data set variable which has as many observations as there are list items

- each parameter is placed in its own data set variable, in a data set which contains a single observation

To illustrate the second technique, consider how it might be applied to our example parameters.

```
data mydata.parms(label=
 'MLD Run #3 on 01Jan1999') ;
  length
   title    $ 040
   dataset  $ 017
   acct001  $ 005
   acct002  $ 005
   acct003  $ 005
   startdt    008
   thrudt     008
   notify   $ 040
   ;
title='GBN-MD Corp. Big Accounts' ;
dataset='mydata.sales' ;
acct001= '00001' ;
acct002= '00002' ;
acct003= '00005' ;
startdt='01JUL1998'd ;
thrudt='31DEC1998'd ;
notify='maddoggy@gbnmd.com' ;
format startdt thrudt date9. ;
label
 title= 'Title'
 dataset= 'Data Set Name'
 acct001= 'Account #1'
 acct002= 'Account #2'
 acct003= 'Account #3'
 startdt= 'Start Date Range'
 thrudt= 'Thru Date Range'
 notify= 'Notify E-Mail Address'
 ;
run ;
```

This approach has some very good features. Anyone with access to SAS software, even if it is only the SAS System Viewer, can examine the data set and see what parameter values are being passed.

The variable labels can be used to provide a more easily understood description of each parameter than is otherwise possible using Version 6 data set names, which are limited to 8 characters. When variable formats, such as the Date9. format used for Start Date and Thru Date, readability is further enhanced.

Unlike macro variables, parameter data sets created in a permanent library exist from one SAS session to the next. Using SAS/CONNECT, transport data sets, and other methods, parameter SAS data sets can be easily moved from one computer to a second one.

Unfortunately, there are downsides to the parameter data set approach. The single observation data set does not handle lists with varying numbers of list items well. Although you could create multiple data sets or tables in

a relational database that could overcome this, this may not be practical. In addition, in Version 6 character strings are limited to 200 characters for both single observation and multiple observation parameter data sets.

How do we apply parameter data sets to successor processes? One way is to convert the data set into a series of macro variables. Consider the following example.

```
data _null_ ;
  set parms ;
  call symput('title',title) ;
  call symput('dataset',dataset) ;
  call symput('acct001',acct001) ;
  call symput('acct002',acct002) ;
  call symput('acct003',acct003) ;
  call symput('startdt', put(startdt,8.)) ;
  call symput('thrudt',  put(thrudt,8.)) ;
  call symput('notify',notify) ;
run ;

%put &title ;
%put &dataset ;
%put &acct001 &acct002 &acct003 ;
%put &startdt &thrudt ;
%put &notify ;
```

The %put statements are provided in this example so the reader can more easily see the results of the SYMPUT calls. Note that numeric parameters, such as Start Date and Thru Date, must be converted to character strings using the PUT function before SYMPUT is called.

Now if parameter lists are being passed as a SAS data set with a separate observation for each list item for the purpose of subsetting a larger SAS data set, there are a few good ways to use the parameters. The parameter data set could be merged with the larger one, passing only those observations where both data sets contributed to the merged observations. A similar filtering can be accomplished using the SQL procedure.

A Note on Formats

However, if the parameter data is small enough to fit into memory, it is hard to beat using a subsetting informat. This technique avoids sorting large data set explicitly or implicitly and gives good performance for nearly all situations. In our example, this is how the technique might be implemented.

```
data accounts ;
  input @1 start $5. ;
datalines ;
00001
00002
00005
;

data cntlin ;
  length
   fmtname $ 008
   start   $ 008
   label     008
   type    $ 001
   hlo     $ 001
   ;
  retain
   fmtname 'acctin'
   label   1
   type    'i'
   hlo     ' '
   ;
  set accounts end=end ;
  output ;
  if end then do ;
    start= '' ;
    label= 0 ;
    hlo= 'O' ;
    output ;
  end ;
run ;

proc format cntlin=cntlin ;
run ;

/* print using subsetting format */
proc print data= mydata.sales ;
   title1 'GBN-MD Corp Big Accounts' ;
   where
     saledate ge '01JUL1998'd and
     saledate le '31DEC1998'd and
     input(account,acctin.) ;
run ;
```

The second data set creates a CNTLIN data set, which is fed to the FORMAT procedure. The format procedure creates the ACCTIN informat, which returns 1 (true) for the three accounts in the ACCOUNTS data set (00001, 00002, 00005) and 0 (false) for any other account. The WHERE statement in the PRINT procedure will only print observation which the INPUT function applied to the Account data set variable yields a value of true and which falls between the selected date range.

Passing Parameters Using Sequential Text Files

All of the parameter passing techniques covered thus far presume that the parameters are being created by and sent to SAS programs. An alternative assumption employed is that where non-SAS program are involved, they are sending or receiving data streams that can be interpreted by SAS software through the use of the appropriate data set engines.

However, we often need techniques that will work with other types of software. For example, when interfacing with SAS running on a mainframe computers in batch mode, Job Control Language (JCL) must surround the SAS language program statements.

Consider the following example. The first step of a mainframe batch job extracts only those records containing our three selected account numbers (00001, 00002, and 00005) are extracted from a tape file and copied to a second tape file using a utility program called SyncSort®. Then in the second step, the tape extract is read by SAS and written to a permanent SAS data set in the libref MYDATA.

```
//G1234X JOB (MYACCT),'MAD DOGGY X1234',
// CLASS=S,MSGCLASS=P,MSGLEVEL=(1,1),
// TIME=(0,1),NOTIFY=MADDOGGY
//STEP1    EXEC PGM=SORT,REGION=1500K
//*-------------------------------------
```

```
//SORTIN  DD DSN=A1234.RAWTAPE,DISP=SHR
//*-------------------------------------
//SORTOUT DD DSN=A1234.INFILE,
//           DISP=(NEW,PASS),
//           UNIT=TAPE,
//           LABEL=(1,SL),
//           VOL=(,,,99)
//*-------------------------------------
//SYSOUT   DD SYSOUT=*
//SYSUDUMP DD SYSOUT=*
//SYSDBOUT DD SYSOUT=*
//SYSIN    DD *
  INCLUDE COND=((1,5,ZD,EQ,00001),OR,
              (1,5,ZD,EQ,00002),OR,
              (1,5,ZD,EQ,00005))
  SORT FIELDS=(200,1,PD,A)
  END
//*-------------------------------------
//STEP2    EXEC SAS
//*-------------------------------------
//IN1      DD DSN=A1234.INFILE,
//            DISP=(OLD,DELETE)
//MYDATA   DD DSN=A1234.MYDATA,
//            DISP=(NEW,CATLG,DELETE),
//            UNIT=DISK

< SAS statements go here >

/*
//
```

Readers interested in learning more about constructing parameter-driven SAS applications on mainframes may wish to consult the two papers previous authored by Michael Davis, which have been cited in the Bibliography. However, the key idea to note is that flat files containing the appropriate control statements can be used to pass parameters between SAS applications and non-SAS programs. SAS programs utilizing DATA _NULL_ steps, macro language, and SCL can write these control statements.

Another reason for using sequential text files for parameter passing is that they can solve the 200-character limit for SAS data set and SCL variables. If a message or character string of a longer length needs to be passed, a sequential text file is a good alternative.

Parameter text files passed to word processing programs and web browsers may first need to be formatted with tags, control codes, or delimiters for successful use. In addition to the techniques previously mentioned, some versions of SAS have "wizards" or SAS/ACCESS interfaces to automate the translation SAS data sets into the stylized files that may be required.

Passing Parameters Using ENTRY Statements

The techniques discussed thus far can be used to pass parameters to SAS/AF catalog entries as well as base SAS programs. However, additional techniques are available for passing parameters to FRAME, PROGRAM, and SCL catalog entries.

Obviously, some parameters can be entered in fields on PROGRAM entry screens or can selected using the widgets on FRAME entry screens. Even when screens are available, it may be necessary to pass some parameters before a catalog entry is started. Parameter passing techniques are also required if the entry is to be run without operator intervention.

One of the most common ways to pass parameters to catalog entries is through the ENTRY statement. The ENTRY statement is similar in concept to positional parameters used with macros. To illustrate, consider the following example. From within a parent catalog entry, the following DISPLAY call is issued to invoke the PRACCTS.SCL entry.

```
call display('praccts.scl',
 dataset,title,startdt,enddt,
 acct001,acct002,acct003,notify) ;
```

As the example illustrates, the first parameter passed in the CALL DISPLAY is the name of the receiving catalog entry. The receiving catalog entry name is followed by parameters, separated by commas. The same syntax applies when parameters are send using a CALL GOTO statement. The receiving catalog entry, PRACCTS.SCL, might look like this.

```
entry
 dataset $ 17 title $ 40
 startdt enddt 8
 acct001 acct002 acct003 $ 5
 notify $ 30
 ;

init:
  submit continue ;
  proc print data= &dataset ;
     title1 "&title" ;
     where account in(&acct001
           &acct002 &acct003) and
           saledate ge &start and
           saledate le &thru ;
  run ;
  endsubmit ;
return ;
```

The syntax for the ENTRY statement is the word "ENTRY", followed by the names of the receiving SCL variables, which are delimited by spaces, not commas. After each variable or group of similar variables, a dollar sign ($) if the variables are character, and the variable's length follows.

When there are no options in the ENTRY statement, the number of parameters, their type, and their order has to match the parameters passed by the CALL DISPLAY or CALL GOTO. OPTION= allows omitted parameters to be initialized to missing values. Otherwise, any mismatch between the parameters supplied by first catalog entry and those listed in the ENTRY statement will cause the target catalog entry to fail to run. The ARGLIST= allows the first program to pass an SCL list to the second program.

Version 7 introduces several additional features to the ENTRY statement. The INPUT argument prevents the parameter from being copied back to the first catalog entry when the second entry is invoked using a CALL DISPLAY statement. The data type may be declared with

the colon (:) modifier. The RETURN= allows some additional flexibility in passing back parameters to the first entry.

However, at the time that this paper is being written, those dealing with parameter passing are probably working within the limits of Version 6. One big advantage of using the ENTRY method for passing parameters is that in SCL, the parameter name length limit is expanded to 32 characters. Thus, a parameter name such as

```
A_VERY_LONG_AND_MEANINGFUL_NAME
```

is permitted. However, the biggest problem with the ENTRY statement technique for passing parameters that remains is the task of keeping the number, type, and order of parameters synchronized between the sending and receiving catalog entries. Using SCL lists to pass parameters overcomes this shortcoming, as we will see.

Environmental SCL Lists

For the uninitiated, SCL lists are ordered collections of data stored in memory and can be accessed during the execution of FRAME, PROGRAM, and SCL catalog entries. For more background information about using SCL lists, please consult the paper by Lisa Ann Horwitz listed in the Bibliography.

Even those familiar with the power of SCL lists may not understand that list items (parameters) may be placed in and retrieved from the global environment SCL list that is automatically created when a SAS session is started. This environment list is analogous to the global symbol table of the macro language.

Also analogous to the macro language, when the AF (or FSEDIT) command is issued a local environment SCL is created. The local environmental SCL list persists for the duration of the AF (or FSP) application that created it.

So how do we put items in this global list? Observe the following example.

```
init:
  /* set up the lists */
  genvlist=envlist('g') ;
  parmlist=makelist(0,'g') ;
  acctlist=makelist(0,'g') ;

  /* Create the items in the list */
  rc=setniteml(genvlist, parmlist,'Parms List');
  rc=setnitemc(parmlist, title,'TITLE') ;
  rc=setnitemc(parmlist, dataset,'DATA SET') ;

  /* Create the sub-list */
  rc=setniteml(parmlist, acctlist, 'ACCOUNT
    LIST') ;
  rc=insertc(acctlist, acct001,-1) ;
  rc=insertc(acctlist, acct002,-1) ;
  rc=insertc(acctlist, acct003,-1) ;
  rc=setnitemn(parmlist, startdt,'START DATE') ;
  rc=setnitemn(parmlist, thrudt,'THRU DATE') ;
  rc=setnitemc(parmlist, notify,'NOTIFY E-MAIL')
    ;
return ;
```

Some subtle features of the preceding SCL code should be noted. First PARMLIST was created and inserted into the global list, GENVLIST. One reason for this is that if multiple applications are run in the current SAS session, there is less opportunity for name collisions than if the parameter items were directly inserted into the global list. Also, using a parameter sublist allows the parameter list to be more easily deleted or cleared when through.

Also note that we can insert and set list items in sublists, even after the lists are placed into the master lists. This is illustrated by placing items in PARMLIST after it has been placed into GENVLIST. Also, we insert accounts into ACCTLIST after it has been placed into PARMLIST. This is because when we use the INSERTL or SETNITEML functions, we are only inserting the list identifier, which is actually only a number, and not the actual list itself.

Last, for parameter lists, it is generally a good idea to use named list items and to use the "SETNITEM" functions (such as SETNITEMC, SETNITEML, and SETNITEMN) instead of the "INSERT" functions. It is much easier to retrieve items by name than position. Also, the "SETNITEM" functions overlay items of the same name, preventing confusion and saving memory.

However, for parameter lists that will contain lists with varying numbers of items, item names are often not necessary. Also they can be place in the appropriate sublist using the INSERTC or INSERTN functions. The author supplies the -1 position parameter to indicate that the item is to be placed at the end of the list.

Were we to inspect our SCL lists using the following PUTLIST call

```
  call putlist(genvlist, 'Global List',1);
```

the information returned by the call would look something like this.

```
Global List( PARMS LIST=(
   TITLE='GBN-MD Corp. Big Accounts'
   DATA SET='mydata.sales'
   ACCOUNT LIST=( '00001'
                  '00002'
                  '00005'
                  [6]
   START DATE=14061
   THRU DATE=14244
   )[4]
)[2]
```

To retrieve the parameters from the SCL list so that we can use them in the second program, the following SCL statements might be employed.

```
init:
  genvlist=envlist('g') ;
  parmlist=getniteml(genvlist, 'PARMS LIST') ;
  title=getnitemc(parmlist, 'TITLE') ;
  dataset=getnitemc(parmlist, 'DATA SET') ;
  acctlist=getniteml(parmlist, 'ACCOUNT LIST') ;
  startdt=getnitemn(parmlist, DATE') ;
```

```
  thrudt=getnitemn(parmlist, 'THRU DATE') ;
  notify=getnitemc(parmlist, 'NOTIFY E-MAIL') ;
  list_len= listlen(acctlist) ;
  do n= 1 to list_len ;
     account=getitemc(acctlist,n) ;
  end ;
return ;
```

When a large number of parameters needs to be passed in a SAS/AF application, or when some of the parameters consist of lists, and the catalog entries share the same environment, environmental SCL lists are a good technique to employ. However, environment SCL lists have a couple of shortcomings that are easily resolved.

SLIST Catalog Entries

Since SCL lists, including the environmental lists, are maintained in memory, when the SAS session ends, the lists disappear. Also, when applications function across platforms, the ability to pass the parameters stored in SCL needs to be addressed.

Fortunately, SCL lists can be saved to SLIST catalog entries. Further, the catalogs in which the SLIST catalog entries are stored can be transported by SAS/CONNECT or other means to another computer. Last, catalogs containing SLIST entries converted to SAS transport format files.

To convert parameter list, PARMLIST to an SLIST entry, we might use the SAVELIST function as shown in the following example.

```
rc=savelist('catalog',
    'mydata.mycat.parmlist.slist' ,0,'My
    Parameter List') ;
```

In this example, the items in PARMLIST are being saved to the PARMLIST.SLIST entry in the catalog MYCAT, which is stored in the MYDATA libref. Since we only need the third parameter (attribute list) as a placeholder, we enter the constant zero. The fourth (last) parameter is the description that appears when we inspect the catalog in the Catalog window.

Then, in the calling program, the SCL list is restored to the PARMLIST for use via the FILLIST function.

```
parmlist=makelist() ;
rc=fillist('catalog',
    'mydata.mycat.parmlist.slist',
      parmlist) ;
```

One might ask, "Can I use SCL lists within base SAS applications?" The answer is yes, in concert with macro variables. Use the DISPLAY procedure or the AF Display Manager command to run an SCL entry that fills an SCL list entry with the contents of an SCL. Then use SYMPUT calls to copy the parameters to macro variables.

Which Technique to Use?

With all the parameter passing techniques covered, the potential for confusion over which technique to use is a real possibility. So the authors offer the following advice on how to select the appropriate technique for a given application.

When passing parameters with Base SAS within the same SAS session, macro variables, combined with the macro language and CALL EXECUTE work very well. To save or move parameters to a second SAS session, parameter SAS data sets with a single observation are a good choice. When lists of varying lengths are required, parameter SAS data sets with multiple observations fit the bill.

For SAS/AF applications with straightforward requirements, the ENTRY statement is a good choice. However, when the list of parameters to be passed is unwieldy, SCL lists and SLIST entries should be the first choice.

For passing parameters from base SAS to SAS/AF applications, consider creating macro variables in base SAS and reading their values with the SYMGET function. For passing parameters the other way, create macro variables and insert the appropriate ampersand (&) prefixed references within the DATA STEPS and PROCedures.

Finally, to pass parameters to programs that cannot read SAS data set, both Base SAS and SAS/AF can create an appropriate flat file with the appropriate tags and delimiters.

PASSING PARAMETERS TO BATCH APPLICATIONS

Up to this point, we have discussed the various methods for handling parameters in SAS programs. Now let's turn to how you handle the execution of those programs when the program is not going to be executed immediately.

The key concept to consider when passing parameters to SAS applications run in batch mode is that a new SAS session is usually opened. In fact, although programs are usually executed in the same environment in batch as in interactive processes, you need to take precaution that the environment and all the things you had available to that SAS session in interactive mode are also available to you in batch. Examples of things that might not always be consistent across environments include:

- the user ID (which may be a generic system ID with different privileges)
- some SAS macro variables
- the SAS configuration (the autoexec.sas and config.sas files may be different)

- some operating system environments could be different such as the local environment variables (this is especially true in UNIX environments) [2]

You want to ensure the macro variable methods and SCL List methods that are used effectively within the same SAS session are going to work. The parameters are going to have to be passed through a parameter SAS data set or via an SLIST catalog entry.

Consider the following example. We have the appropriate parameter data set and wish to run our PRINT procedure in batch on an Microsoft Windows computer. We might launch our batch program in the following manner.

```
C:\sas\sas.exe -sysin C:\sas\mylib\pracct.sas
  -config C:\sas\config.sas
```

Please note that the preceding command is entered on a single line. Unless we supply additional parameters to this command, the SAS Log is written to c:\sas\mylib\pracct.log and the output from the PRINT procedure is written to c:\sas\mylib\pracct.lst.

In the UNIX environment, you can run SAS jobs in batch using the following method

```
sas pracct.sas -config ~/myconfig.sas &
```

where "sas" points to a shell script or the actual executable for your UNIX machine.

Please refer to the specific operating system companion guides (for SAS) for information on how to run SAS in batch (or non-interactive) mode.

Conditional Execution

Often when we execute SAS jobs in batch, we need to be concerned with how we conditionally execute sections of a batch process. Also, we may need the ability to restrict the execution of a section of code when the previous job has failed.

One technique is to use the EXECUTE call to run a macro only when the previous step has completed successfully. A good way to accomplish this is have a successful step write a parameter data set with the completion information.

Scripting

Scripting is the ability for the operating system to run a defined set of instructions. These can either be run interactively or via a scheduler. The examples given above might be put into a separate file and a script can then be responsible for running the program and conditionally executing based off of return codes.

[2] For a more complete discussion of this in the UNIX environment please refer to the paper by Barnes Nelson on using shell scripts and SAS cited in the Bibliography.

In the Windows NT environment, you can take advantage of batch (CMD) files that allow sequential execution of commands. In addition simple tests/logic operators can be added.

In the UNIX environment "Shell" scripts allow execution of commands and offer powerful programming functions such as if/then/else, case statements, functions, redirection and others. In fact, the majority of system startup/configuration files are written using "Bourne" Shell Scripts.

Both platforms support implementations of other "script-type" languages such as Perl, which can be a great aid in system management.

In Microsoft Windows 95/ 98 environments, you may also be able to make use of the Task Scheduler. In Win 98 this is native, in Win 95 this is an add-on.

Automatic Notification of Job Status

In addition, we might we send an e-mail message upon completion or failure of a batch job. We could use the ability of SAS to write flat files to write to the SMTP (Simple Mail Transport Protocol) receiver using the SOCKET access method or by using the PIPE filename option in the UNIX environment.

For example, the following program can be executed to communicate with the UNIX mail program.

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

Please see the paper by Jack Shoemaker referenced in the Bibliography for more details on using the SOCKET access method.

One final caveat to keep in mind when sending completion messages from a batch SAS job is that if a step does not complete successfully, it probably will not complete sending an end of job message. So the message step should be engineered as a separate step that looks for a parameter data set or flat file and sends an exception message if the data set or file is not found.

Web-Enabled SAS Applications

Web applications are a special case of running batch jobs that deserve comment. Similar to the techniques we used to pass parameters to SAS applications run in batch, we can also use these to pass information from an HTML web form to a subsequent program (here we will call either a SAS program or an SCL program). In addition to the previously covered techniques, the script

program referenced by the web page allows us to pass additional parameters set by the user's screen selections. Using an Attribute <A> HTML tag, the syntax is.

```
<A HREF="CGI-program-address?parameters>
```

If our program were web-enabled, the HTML tag might look something like the following for a Base SAS or macro program

```
<A HREF="/cgibin/broker?_service=
default&_program=wcgi.pracct.sas
&account=00001">
```

Similarly, we can call an SCL program.

```
<A HREF="/cgi-bin/
  broker.exe?_PROGRAM=sample.logon.scl
  &_service=default
  &userid=greg
  &password=ha
  &reqtype=init
```

Multiple parameters are often passed using HTML forms.

PASSING PARAMETERS TO SCHEDULERS

The techniques covered in the previous section on how to pass parameters to SAS tasks run in batch also apply to jobs submitted by schedulers. A scheduler is a program that is always running in background and submits a task at the scheduled times.

Earlier, we spoke about scheduling large jobs to run off-hours. A host scheduler allows “batch” jobs to be run at certain intervals unattended. Systems typically have a low-usage time and by using a scheduler jobs that would normally run during “peak” hours can be batched to run at night. In addition schedulers can be used to implement accounting, system checks, backups, database refreshes etc.

Depending upon the platform (OS) on which SAS is run, native schedulers (supplied by the OS vendor) may be available. If not, schedulers may licensed from third parties or may be supplied as part of another utility.

In the Windows NT environment, you have access to the AT command as well as third-party schedulers (e.g., Argent Software http://www.argent-nt.com).

In the UNIX environment, you may have access to the AT command, the crontab facility (Chronological timer) as well as third-party schedulers. Cron is probably the most flexible and can be driven from external programs such as SAS.

CONCLUSION

The authors hope that the techniques outlined in this paper enable others to automate their SAS applications and allow them to delegate to others the mundane task of running them. We expect that the readers of this paper will now be in a position devote more time to their local SAS user groups and to join us at SAS conferences.

Please note that this paper is still being revised and improved. A link to the most current version in Adobe Acrobat (PDF) has been posted to the World Wide Web. A link to it can be found at the URL

http://www.bassettconsulting.com.

BIBLIOGRAPHY

Barnes-Nelson, G.S. (1996) "An Introduction to UNIX Shell Programming and the SAS System," *Proceedings of the Twenty-First Annual SAS Users Group International Conference*.

Barnes-Nelson, G.S. (1999) "Extending the Life of Your AF Application: Exploiting the Model-Viewer Paradigm," *Proceedings of the Twenty-Fourth Annual SAS Users Group International Conference*.

Davis, Michael (1996), "FRAME Your Mainframe Batch Applications, " *Proceedings of the Twenty-First Annual SAS Users Group International Conference,* 21, 1223-32.

Horwitz, Lisa Ann (1998), "Harnessing the Power of SCL Lists," *Proceedings of the Twenty-Third Annual SAS Users Group International Conference,* 23, 48-56.

Shoemaker, Jack (1997) "Let's Not Forget E-Mail", *Proceedings of the Twenty-Second Annual SAS Users Group International Conference,* 22, 878-83.

Terr, Adam and Davis, Michael L. (1998), "Driving to Better Credit Policies : The Risk Strategy Instrument Panel," *Proceedings of the Eleventh Annual NorthEast SAS Users Group Conference,* 11, 186-93.

CONTACT INFORMATION

The authors may be contacted as follows:

Michael L. Davis
Bassett Consulting Services, Inc.
10 Pleasant Drive
North Haven CT 06473-3712
Internet: michael@bassettconsulting.com
Web: http://www.bassettconsulting.com
Telephone: (203) 562-0640
Facsimile: (203) 498-1414

Gregory S. Barnes Nelson
STATPROBE, inc.
104 Sterling Ridge Way
Morrisville, NC 27560
Internet: greg.barnesnelson@statprobe.com
Web: http://www.statprobe.com