# Calling External Modules from the SAS® System

Richard D. Langston, SAS Institute Inc.


## ABSTRACT

The new MODULE routines allow for invoking external subroutines written in COBOL, C, assembler, and other languages within DATA steps, SAS/IML® steps, and SCL steps. This paper introduces the reader to the MODULE routines. These routines are an outgrowth of the COBOLINT routine that has been discussed in previous articles.


## INTRODUCTION

There have been many requests over time for the SAS user to have the ability to call external subroutines from within the SAS System. The ability has always been there via the X command for invoking an external command, or via the PROC statement to invoke an external program. However, users have external subroutines that need to be called with argument lists. Also, these routines may wish to return values and/or update the arguments. These need is beyond the scope of the X command or the PROC statement.

The first attempt at providing the routine invocation ability was through the COBOLINT routine, which accesses COBOL subroutines. The COBOLINT routine was described in *Observations 1993Q1* and has been provided to several test sites. However, as the testing evolved, it was realized that routines in other languages needed to be accessed. Also, it was realized that SAS/IML Software would be an ideal environment to allow calling of subroutines that use vectors and matrices in their calling sequence. These additional needs resulted in a new version of COBOLINT, expanded into several routines with the prefix MODULE.

## BACKGROUND ON COBOL INTERFACING

A unique feature of COBOL subroutines is their ability to be invoked from a non-COBOL environment. The subroutine will establish the necessary COBOL environment, and run to completion, as long as it is provided the proper parameters. Because of this dynamic loading feature, users find it desirable to be able to invoke these dynamically loadable modules from within the DATA step environment.

Consider a simple example. The following COBOL routine accepts four arguments. Each argument is of a different data type. Each argument's value is changed by incrementing by 1.

```
IDENTIFICATION DIVISION.
PROGRAM-ID.    DECUSED.
AUTHOR.        RICHARD D. LANGSTON

ENVIRONMENT DIVISION.

DATA DIVISION.
WORKING-STORAGE SECTION.

LINKAGE SECTION.
 01 ZD4-1     PIC S999V9 DISPLAY.
 01 PD4-1     PIC 99999V9 PACKED-DECIMAL.
 01 IB2-1     PIC S999V9 BINARY.
 01 PIC4-1    PIC 999V9.

PROCEDURE DIVISION USING ZD4-1, PD4-1,
                         IB2-1, PIC4-1.
   ADD 1 TO ZD4-1  GIVING ZD4-1.
   ADD 1 TO PD4-1  GIVING PD4-1.
   ADD 1 TO IB2-1  GIVING IB2-1.
   ADD 1 TO PIC4-1 GIVING PIC4-1.
   GOBACK.

END PROGRAM DECUSED.
```

From COBOL, the above COBOL subroutine could be called as follows:

```
....
01 ARG1      PIC S999V9 DISPLAY.
01 ARG2      PIC 99999V9 PACKED-DECIMAL.
01 ARG3      PIC S999V9 BINARY.
01 ARG4      PIC 999V9.
....
   MOVE 1 TO ARG1.
   MOVE 2 TO ARG2.
   MOVE 3 TO ARG3.
   MOVE 4 TO ARG4.
   CALL "DECUSED" USING A, B, C, D.
....
```

After the call to DECUSED, the values of ARG1, ARG2, ARG3, and ARG4 would be 2, 3, 4, and 5, respectively.

What COBOL and SAS users would like is to be able to call the routine in a similar fashion in the DATA step:

```
DATA _NULL_;
     ARG1=1; ARG2=2; ARG3=3; ARG4=4;
     CALL DECUSED(ARG1,ARG2,ARG3,ARG4);
     PUT ARG1= ARG2= ARG3= ARG4=;
```

Where ARG1-ARG4 would have the values of 2, 3, 4, and 5, respectively.

This seems like a simple request, but it is actually very complicated. The main difficulty lies in the fact that the DATA step cannot determine the characteristics of the arguments that DECUSED expects. In the DECUSED example above, the first argument is zoned decimal with 1 place to the right of the decimal. Argument 2 is packed decimal, also with 1 place to the right of the decimal. Unless the DATA step function can convert the floating point arguments into the proper binary streams to pass to the COBOL program, DECUSED cannot function correctly.

Of course, if the DATA step user bears the burden of converting the values to the proper binary streams, then the problem is much less difficult:

```
DATA _NULL_;
     ARG1=1; ARG2=2; ARG3=3; ARG4=4;
     XARG1=PUT(1,ZD4.1);
     XARG2=PUT(1,PD4.1);
     XARG3=PUT(1,IB2.1);
     XARG4=PUT(1*10,4.);
     CALL DECUSED(XARG1,XARG2,
                  XARG3,XARG4);
     ARG1=INPUT(XARG1,ZD4.1);
     ARG2=INPUT(XARG2,PD4.1);
     ARG3=INPUT(XARG3,IB2.1);
     ARG4=INPUT(XARG4,4.1);
```

```
PUT ARG1= ARG2= ARG3= ARG4=;
RUN;
```

This burden is acceptable to some users, and not to others. For those users who find this unacceptable, a method to automatically convert the values is desired.

### PROPOSED SOLUTION

The proposed solution to the problem is the introduction of the MODULE routine. This routine is responsible for accepting an external routine name and a series of arbitrary arguments. The routine name is looked up in an optional attribute table that can be supplied by the user. The table contains all the attributes for the arguments. MODULE is responsible for converting the user's arguments into the proper format to be passed to the external routine. The external routine is dynamically loaded and invoked with the proper calling sequence. Once the external routine returns, MODULE converts the arguments back into the proper representation to return to the DATA step. Using our example above, modified to use the MODULE routine:

```
DATA _NULL_;
     ARG1=1; ARG2=2; ARG3=3; ARG4=4;
     CALL MODULE('DECUSED',
                 ARG1,ARG2,ARG3,ARG4);
     PUT ARG1= ARG2= ARG3= ARG4=;
     RUN;
```

The values of ARG1-ARG4 would be 2, 3, 4, and 5 respectively.

The attribute table is a file referred to by the SASCBTBL fileref, which is optionally given. Here is the attribute table that would be used in conjunction with DECUSED:

```
routine decused minarg=4 maxarg=4;
arg 1 update format=zd4.1;
arg 2 update format=pd4.1;
arg 3 update format=ib2.1;
arg 4 update format=4.1;
```

To allow for the user who does not wish to use the attribute table, the MODULE routine allows calls to an external routine without using a table. Here is a revision of the SAS program given above, calling the DECUSED module without an attribute table:

```
DATA _NULL_;
     ARG1=1; ARG2=2; ARG3=3; ARG4=4;
     XARG1=PUT(1,ZD4.1);
     XARG2=PUT(1,PD4.1);
     XARG3=PUT(1,IB2.1);
     XARG4=PUT(1*10,4.);
     CALL MODULE('DECUSED',XARG1,XARG2,
                           XARG3,XARG4);
     ARG1=INPUT(XARG1,ZD4.1);
     ARG2=INPUT(XARG2,PD4.1);
     ARG3=INPUT(XARG3,IB2.1);
     ARG4=INPUT(XARG4,4.1);
     PUT ARG1= ARG2= ARG3= ARG4=;
     RUN;
```

The values of ARG1-ARG4 would be 2, 3, 4, and 5 respectively, as they were in the previous example.

This method may actually by preferred by many users, since many COBOL users, for example, pass their arguments as character strings or simple numeric pictures, which are easily handled via character variables.

What has been described so far concerns each MODULE argument (after the module name) corresponding to the same argument being passed to the external routine. What users find also useful is to pass a block of parameters to correspond to one or more blocks of data, such as COBOL field definitions. Consider this COBOL program:

```
IDENTIFICATION DIVISION.
PROGRAM-ID.    FDTEST.

ENVIRONMENT DIVISION.

DATA DIVISION.

LINKAGE SECTION.
 01 FD-1.
    10 KEY-1 PIC X(10).
    10 AGE-1 PIC 999 DISPLAY.
    10 NAME-1 PIC X(20).
 01 FD-2.
    10 SEX-1 PIC X.
    10 BDATE-1 PIC X(6).
    10 VALUE-1 PIC 9999 DISPLAY.
PROCEDURE DIVISION USING FD-1, FD-2.
    IF KEY-1 = "ABCDEFGHIJ"
       THEN PERFORM GET-ABCDEFGHIJ
    ELSE IF KEY-1 = "1234567890"
       THEN PERFORM GET-1234567890.
    GOBACK.

  GET-ABCDEFGHIJ.
    MOVE 25 TO AGE-1
    MOVE "JANE JONES" TO NAME-1
    MOVE "F" TO SEX-1
    MOVE "102767" TO BDATE-1
    MOVE 133 TO VALUE-1
    GOBACK.

  GET-1234567890.
    MOVE 38 TO AGE-1
    MOVE "RICK LANGSTON" TO NAME-1
    MOVE "M" TO SEX-1
    MOVE "031955" TO BDATE-1
    MOVE 427 TO VALUE-1.
    GOBACK.

END PROGRAM FDTEST.
```

This COBOL program expects to be passed two parameters, both of which are field definitions. A key (KEY-1) is passed as part of the first field definition. Based on that key value, the remaining values of both field definitions are filled in. This is similar to what a "real-life" COBOL application would do: look for data corresponding to a key (via a key in a file or a database) and map that data record back into the field definition. The above example simply copies in data based on two different key values, but the method for obtaining the proper data is totally irrelevant to MODULE.

At this point we introduce the concept of the *control string* argument to MODULE. If the first argument has an asterisk (*) as its first character, then MODULE recognizes the argument not as a routine name but as a list of options. The second argument is then expected to contain the routine name. There are several different option values that can appear in the control string, and the pertinent value in this discussion is the *S* option.

The above FDTEST program would be invoked using MODULE with the S option in the control string to indicate that the arguments being passed are divided into field definitions:

```
DATA _NULL_;
     LENGTH KEY $10;
     LENGTH AGE $3 VALUE $4 NAME $20
            SEX $1 BDATE $6;
     KEY='ABCDEFGHIJ';
     CALL COBOLINT('*S/','FDTEST','/',KEY,
                   AGE,NAME,
                   '/',SEX,BDATE,VALUE);
     VALUEX=INPUT(VALUE,ZD4.);
     PUT _ALL_;
     RUN;
```

The above code produces the following on the log:

```
KEY=ABCDEFGHIJ AGE=025 VALUE=0133
NAME=JANE JONES SEX=F
BDATE=102767  VALUEX=133
```

(The above code assumes we are NOT using an attribute table entry for FDTEST).

The / character after the S option in the control string indicates the separator character that delimits the list of arguments for each field definition. The / character should appear as a separate argument that is one character long, either as a constant or as a LENGTH $1 variable. You should choose your separator character such that no valid argument could be that one-byte character value.

## BEYOND THE COBOL ARENA

As testing of the MODULE function predecessor, COBOLINT, proceeded, it was realized that users needed access to routines other than just COBOL routines. There were subroutine libraries written in C, for example, that could be accessed in a manner similar to that discussed for COBOL. Moreover, there was a need to access C and FORTRAN routines that would refer to arguments that were not just scalar values, including vectors and matrices.

Once C was considered, one had to contend with return codes. The COBOL language does not consistently support a return code, but return arguments are a part of the native C language, as they are for FORTRAN and PL/I. This meant the introduction of more than just the MODULE CALL routine. The new routines were called MODULEN, a function returning a numeric argument, and MODULEC, a function returning a character argument. The arguments to MODULEN and MODULEC are the same as those to the MODULE CALL routine.

Of course, this meant an extension to the attribute table, so that the type of return argument could be described. This argument would have to be converted similarly to the standard arguments passed in the calling sequence. And care was especially necessary for returning character arguments, since the definition of character data differs between the SAS System and languages such as C. C, in fact, does not portably return character data (except as single byte values cast to ints). A C function can return a pointer to character data, so the MODULEC routine assumes that the SAS user provides a sufficiently large character variable to receive the returning bytes. For example, if the C function XYZ returns a pointer to 20 bytes of data:

```
data _null_;
     length x $20;
     x = modulec('XYZ',1,2,3);
     put x=;
     run;
```

The MODULEC function will recognize that the return argument has a maximum length of 20, and will copy the 20 bytes pointed to by the return pointer of XYZ into the proper location for X. An attribute table entry can specify the expected width to allow MODULEC to pad with blanks or truncate appropriately.

## BEYOND THE DATA STEP AND SCALARS

All of the discussion so far has pertained to the DATA step. But the MODULE routine can also be invoked from within an SCL program, just like most other DATA step functions. Also, the MODULE routine can be invoked from within SAS/IML Software, but the elements passed to the MODULE routine and on to the desired subroutine can only be scalar arguments.

Many C and FORTRAN functions expect pointers to vectors and/or matrices. Argument passing works the same way as for scalar values, except that pointers are passed instead of the actual element values themselves. Therefore, the interfacing issues for the MODULE subroutines and functions would not be any more complex, except that the DATA step cannot accommodate non-scalar arguments. However, SAS/IML Software was specifically designed to support matrices. All arguments to SAS/IML functions are considered matrices. Scalars are 1X1 matrices, and vectors are 1Xn matrices.

To support matrices within SAS/IML Software, three new routines were developed: MODULEI, as a CALL routine; MODULEIN, for returning a numeric matrix, and MODULEIC, for returning a character matrix. All three new routines support any number of matrices of either numeric or character type.

The new routines also recognize an attribute table, applying the information to all elements of the matrix. If, for example, the attribute table indicates that a certain argument is an integer, all elements of the matrix are converted to integer values. The proper routine is invoked, then all elements of the matrix are converted back to double-precision numbers. Likewise, if certain character lengths are expected, a new matrix is created with the proper lengths, padding or truncating as necessary.

## HOST-SPECIFIC DISCUSSION

The first work on COBOLINT, and later MODULE, took place on MVS/ESA. This was because all the demand from users for the functionality was for the MVS environment. However, as development progressed, the implementations were made more portable, so that there was no longer the necessity to assume an MVS environment, nor even an IBM mainframe environment.

But when the MODULE functions and subroutines need to invoke external subroutines, host-dependent issues must be considered. The primary concern is that subroutines are not main programs, and are not usually considered executable programs on their own. They may reside only as object code in a library, with the intention that the object code be linked into an executable module at the proper time.

The MODULE functions and subroutines expect to dynamically load an executable program. The executable program is not expected to be invoked in a standalone fashion (such as with EXEC PGM=*program* in MVS JCL or RUN *program* on VAX/VMS), but it must be loadable by the system loader. If the program resides only in object format, it is likely that the system loader will reject the attempt at loading.

On VM/CMS platforms, the executable modules must reside in a LOADLIB file. One uses the LKED command to link objects, CMS

MODULE files, or other LOADLIB members into a new LOADLIB member.

On the VMS platforms, MODULE is restricted to loading executable images. However, these executable images can be the result of a LINK command against an object file that has no MAIN environment. The environment will be proper established automatically when the system loader loads the executable. This applies equally well to COBOL, C, FORTRAN, and PL/I objects. The only requirement for invoking the programs from MODULE is that they be LINKed without unresolved references.

On UNIX platforms, objects must be linked into executables without unresolved references. Like on VMS, it is not necessary for the object code to contain a MAIN. But if the routine is contained only as a "loose" object (a .o file) or as a member of an archive library (a .a file), the routine will have to be linked via the *ld* command into a proper executable image in order for MODULE to load it successfully.

On PC platforms, objects must be linked into proper DLL files before they can be dynamically loaded by MODULE. Moreover, the DLL file must have specialized characteristics so that it can be properly recognized and threaded by the loader invoked by MODULE. The additional tools to create such DLL files are provided as part of the adjunctive software for the MODULE subroutines and functions.

## CONCLUSION

The MODULE suite (MODULE, MODULEN, MODULEC, MODULEI, MODULEIN, and MODULEIC) allow for powerful interfacing to external routines that could not otherwise be invoked by such facilities as the DATA step, SAS/IML Software, and SCL. The connectivity between the SAS language and external subroutine libraries bridges a gap that has often prevented potential SAS applications from being written. With the MODULE suite, developers can have a choice to use the SAS System instead of using only COBOL, C, FORTRAN and other such languages.

## AVAILABILITY

The MODULE suite is planned to be released during Version 6. What particular release depends on the particular host platform. At the time of publication, the MODULE suite is expected to be released as part of 6.08 maintenance for MVS, CMS, and VMS. The suite is expected for the next Version 6 release for UNIX and PC platforms.

## REFERENCES

"The COBOLINT Routine: A Method for Calling COBOL Subroutines from DATA steps." Richard D. Langston, *Observations* 1993Q1, Volume 3, Number 1, pp. 32-37.

SAS, SAS/TOOLKIT, and SAS/IML are registered trademarks of SAS Institute Inc. in the USA and other countries. ® indicates USA registration.