Paper 115-2008

Experiences in Testing Host-Dependent Software in a Portable Fashion

Rick Langston, SAS Institute Inc., Cary, NC

ABSTRACT

This paper describes my experiences with testing MODULE, which invokes executable images on each SAS® host. There were challenges in developing a testing harness being that the creating of these images varies greatly per host. Using innovative techniques, I was able to create portable tests that would produce consistent benchmarks for simplified verification. This paper will describe the testing harness with its various interesting attributes.

THE MODULE ROUTINES: OUR TESTING DILEMMA

The MODULE routines are special functions and CALL routines that can be invoked from any function calling environment, including the DATA step, the macro language, SQL, SCL, and SAS/IML® software. These special functions (hereafter called MODULE for simplicity) call into external subroutine libraries. On Windows, these libraries would be DLLs. On UNIX platforms, these would be shared object libraries. On VMS, they would reside in shareable images. On z/OS, they could be (although not limited to) COBOL subroutine libraries.

MODULE can be tested on Windows by calling into standard KERNEL32 API calls such as MessageBox and Beep. There are also standard routines available on UNIX. But in order to properly test all the different types of arguments and calling sequences, and in order to ensure these are tested across all platforms, it is necessary to write the subroutines that MODULE will call. These subroutines can be written in the C language and can be compiled on all platforms.

What we needed for adequate testing was a front-end SAS program that could be invoked on any platform. The SAS program would invoke macros that would produce the necessary image that the program needed (DLL, shared object, and so on.) We wanted all the testing software to be in the format of SAS programs with no binary data.

We found that we needed to divide the testing process into two parts: 1) creation of images to be used, storing them in a special hex representation, and 2) re-creation of these images into the proper binary form so that they could be invoked by MODULE in the test code.

STARTING AT THE END: THE TEST PROGRAM

It will be clearer to start with one of our test programs. In this example, we want to test calling into external routines that actually increment an internal counter. If the external routine were reloaded with each call, the counter would be reset and would therefore not produce the proper final answer. We wanted to ensure that reloading did not take place.

```
/*************************************************************/
/*          S A S     T E S T     L I B R A R Y                */
/*                                                           */
/*    NAME: counter                                          */
/*   TITLE: Test with static counter in routine              */
/* PRODUCT: BASE                                             */
/*  SYSTEM: ALL                                              */
/*    KEYS:                                                  */
/*   PROCS:                                                  */
/*    DATA:                                                  */
/*                                                           */
/* SUPPORT: Chambers, M.               UPDATE:               */
/*                                                           */
/* This test uses a static counter. This means that the module  */
/* being called is non-reentrant, but some users might want to  */
/* do this.                                                  */
/*************************************************************/

*-----define pertinent macros for this test-----*;
```

```
%let source=counter;
%let entries=counter;
%let prefix3=ctr;

*-----define the SOURCE fileref-----*;
%source_fileref;

*-----write out the C code to the source file-----*;
data _null_; file source;
     input; put _infile_; cards4;
long counter()
{
static long incrementer = 0;
return(++incrementer);
}
;;;;

*-----write out the routine definitions to the SASCBTBL file-----*;
filename sascbtbl temp;
data _null_; file sascbtbl;
     input; put _infile_; cards4;
routine counter minarg=0 maxarg=0 returns=long;
;;;;

 /*-------------------------------------------------------------*/
 /* At this point the %create_image macro is invoked in order  */
 /* to create an executable image for MODULE to use.           */
 /*-------------------------------------------------------------*/

%create_image;

%let infoparm=i;
*%let infoparm=;

 /*-------------------------------------------------------------*/
 /* Test invoking the counter function 10 times. The returned  */
 /* value should increment with each invocation.               */
 /*-------------------------------------------------------------*/

data _null_;
     do i=1 to 10;
        value = modulen("*&infoparm.",'counter');
        put i= value= ' (should be equal)';
        end;
     run;
```

In this test program, we actually provide the C source code for the function we want to call. We also provide the attributes for calling the function (through the SASCBTBL file). Note that this file does not have to be a permanent file, and can instead be a temporary file, as we show here through the use of TEMP. The %CREATE_IMAGE macro will do whatever it takes to produce the image called ‘counter’ so that we can invoke it through MODULE in the subsequent DATA step. Note that the first argument to MODULEN is the information parm, which might be blank, or it might be *I so that additional information is displayed while the test is debugged. Eventually this test was run and its log was saved as a benchmark so that later test runs could be compared to ensure MODULE continued to run properly.

This test was used in two phases: 1) creating the binary image for each platform and producing a hex stream, and 2) regenerating the binary image from the hex stream and running the test. As the developer of the test, I would be the one running phase 1. I would then be able to produce a set of hex streams that could be used on the various platforms so that this test could be run on all platforms.

CREATING THE BINARY IMAGE

The binary image is created by compiling the C code and by linking the C code to a binary image. This process differs from platform to platform. In fact, the ultimate platform on which the testing takes place might not even have a compiler available. This is another reason why it's best to have created these images ahead of time on a machine where the compiler exists. Consider, for example, the Windows platform. It is not typical that a C compiler is present there. As the developer of the tests, I would be using Visual C++ that is installed on my Windows machine to create the images, but once created, the images could be used on any other 32-bit Windows machine. The Visual C++ command I would use to compile and link the DLL for Windows would be the following:

```
CL /DLL /MD counter.c /LINK /EXPORT:counter /OUT:counter.dll
```

The counter.c file would have been obtained from the inline C code in the test.

This same approach would be used for all the other platforms, using the appropriate commands to compile and link the C code.

CREATING THE HEX STREAM

The test program contains the C code needed for compilation. The %CREATE_IMAGE macro has the necessary information to perform the compilation and linking of a binary image based on the platform. Ultimately that binary image (a DLL on Windows machines; a shared object on UNIX) is created, and it needs to be converted to a textual hex stream for simpler maintenance in test libraries. All tests have a 3-character prefix, and all platforms have a 3-character unique abbreviation, so all hex streams can be uniquely named. In the case of the preceding counter test, the prefix is CTR. So our hex stream for 32-bit Windows is ctrhxwin.SAS; for Solaris it is ctrhxs64.SAS, and so on..

Here is the SAS code that will convert a Windows DLL to a hex stream. Note that it is invoked with SYSPARM being set to 'image,prefix,osname'. In our counter example for 32-bit Windows, SYSPARM would be 'counter,ctr,WIN'.

```
data _null_;
     if scan("&sysparm.",3,',')='W64'
        then which='w64';
     else if scan("&sysparm.",3,',')='WX6'
        then which='wx6';
     else which='win';
     dllfile=trim(scan("&sysparm.",1,','))||'.DLL';
     call symput('dllfile',trim(dllfile));
     hexfile=trim(scan("&sysparm.",2,','))||'hx'||which||'.sas';
     call symput('hexfile',trim(hexfile));
     run;
data temp; infile "&dllfile." recfm=f lrecl=1 end=eof;
     length text36 $36;
     retain text36;
     input x $char1.;
     l+1;
     substr(text36,l,1)=x;
     if l=36 or eof then do;
        output;
        text36=' ';
        l=0;
        end;
     keep text36 l;
     run;
data _null_;
     infile cards length=l;
     file "&hexfile.";
     input @; input @1 line $varying200. l;
     i=index(line,'&hexfile');
     if i then substr(line,i,12)="&hexfile.";
     i=index(line,'&dllfile');
     if i then substr(line,i,12)="&dllfile.";
     put @1 line $varying200. l;
```

```
     cards4;
 /*****************************************************************/
 /*          S A S     T E S T     L I B R A R Y                  */
 /*                                                               */
 /*    NAME: &hexfile                                             */
 /*   TITLE: &dllfile    generated from hex data                  */
 /* PRODUCT: BASE                                                 */
 /*  SYSTEM: WIN, DNT                                             */
 /*    KEYS:                                                      */
 /*   PROCS:                                                      */
 /*    DATA:                                                      */
 /*                                                               */
 /* SUPPORT: Chambers, M.                UPDATE:                  */
 /*                                                               */
 /* This SAS code will recreate the &dllfile       from hex data */
 /* so that it can be used with its corresponding test.           */
 /*****************************************************************/

data _null_; infile cards length=l;
             file "&dllfile    " recfm=f lrecl=1;
     input @; input @1 line $varying200. l;
     if line=:'<repeat' then do;
        repeat=input(scan(line,2,'< >'),best12.);
        input @; input @1 line $varying200. l;
        end;
     else repeat=1;
     l2=length(line)/2;
     text=input(line,$hex72.);
     do i=1 to repeat;
        put text $varying36. l2;
        end;
     cards;
;;;;

data _null_; set temp; by text36 l notsorted;
     file "&hexfile." mod;
     if first.l then count=0;
     count+1;
     if last.l;
     if count>1 then put '<repeat ' count '>';
     length hex $72;
     hex=put(text36,$hex72.);
     l2=l*2;
     put hex $varying72. l2;
     run;

data _null_; file "&hexfile." mod;
     put ';' / 'run;';
     run;
```

Most of what is done in the preceding code is creating a standard SAS program that only differs in its instream data. The binary stream is broken into 36-byte pieces to be “hexified” into 72-byte text streams. 36/72 was chosen for the benefit of storing test code on z/OS within a PDS that uses a RECFM=FB and LRECL=80 attribute, and enabling columns 73-80 to be line numbers. Although a limit of 72 isn’t really necessary apart from z/OS, it is simpler to use the same hexification mechanism on all platforms.

Here is the resultant test program that was created from the preceding code, and specifically created for 32-bit Windows:

```
 /*****************************************************************/
 /*          S A S     T E S T     L I B R A R Y                  */
 /*                                                               */
 /*    NAME: ctrhxwin.sas                                         */
 /*   TITLE: counter.DLL generated from hex data                  */
 /* PRODUCT: BASE                                                 */
 /*  SYSTEM: WIN,DNT                                              */
 /*    KEYS:                                                      */
 /*   PROCS:                                                      */
 /*    DATA:                                                      */
 /*                                                               */
 /* SUPPORT: Chambers, M.               UPDATE:                   */
 /*                                                               */
 /* This SAS code will recreate the counter.DLL    from hex data */
 /* so that it can be used with its corresponding test.          */
 /*****************************************************************/

data _null_; infile cards length=1;
             file "counter.DLL " recfm=f lrecl=1;
     input @; input @1 line $varying200. l;
     if line=:'<repeat' then do;
        repeat=input(scan(line,2,'< >'),best12.);
        input @; input @1 line $varying200. l;
        end;
     else repeat=1;
     l2=length(line)/2;
     text=input(line,$hex72.);
     do i=1 to repeat;
        put text $varying36. l2;
        end;
     cards;
4D5A90000300000004000000FFFF0000B800000000000000400000000000000000000000
00000000000000000000000000000000000000000000000D80000000E1FBA0E00B409CD
...
00000000400000C02E72656C6F630000F80300000060000000100000006000000000000000
0000000000000000400000420000000000000000000000000000000000000000000000000000
<repeat 95 >
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000558BECA1F0510010
83C001A3F0510010A1F05100105DC3837C240801757EFF15044000106A00A314520010E8
(many lines skipped here)
...
94309C30A430AC30B430BC30C430CC3000000000000000000000000000000000000000000000000
<repeat 89 >
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000
;
run;
```

The DLL is created in the current directory. The –PATH option is used when running the test, and it refers to the current directory. By the time we run the MODULE step to refer to the DLL, the DLL is already created and is ready to use.

So what we've seen so far is the counter.sas test program used on all platforms, and ctrhxwin.sas, which is the specific code included by %CREATE_IMAGE to create the DLL for the test. There is a corresponding ctrhx*.sas file for every platform being tested: ctrhxs64.sas for Solaris, ctrhxr64.sas for AIX, and so on.. The counter.sas program and all the ctrhx*.sas programs together form the test for confirming static counters.

Z/OS CONSIDERATIONS

Using the technique of hex streams is also feasible on z/OS, but it is a bit more complicated because the executable images are stored in a special form, not just as binary streams as on UNIX and Windows. The executable images used on z/OS are load module members and they must be created with special attributes. This can be accomplished by using the PDSCOPY procedure. PROC PDSCOPY has the OUTTAPE option that is intended to convert a load module to a sequential file for archiving purposes. This file can be hexified. When the load module needs to be re-created, the hex stream can be dehexified back into a PDSCOPY sequential file, and then PROC PDSCOPY can be run with the INTAPE option to convert the stream back to a load module. Once in load module form, MODULE can access the image and run the test.

The MVS dehexification code looks similar to the code for other platforms. However, the difference is that we add a length value so that we can create the records in the VB file with the proper record length. This is important because PROC PDSCOPY needs the records in the precise form so that the load module can be reproduced. Note also that PROC PDSCOPY re-creates the load module in the library referred to by SASLIB. That is a special DDname for z/OS SAS that will be searched prior to the standard installed libraries, making it the ideal place to store our re-created load modules.

```
FILENAME TOPDSCPY TEMP;
DATA _NULL_;
     infile cards;
     FILE TOPDSCPY RECFM=VB LRECL=32756 BLKSIZE=32760;
     INPUT L;
     DO I=1 TO L BY 36;
        INPUT TEXT $HEX72.;
        L2=MIN(36,L-I+1);
        PUT TEXT $VARYING36. L2 @;
        END;
     PUT;
     CARDS;
101
D7C000007FE600000000F94BF1F07F7F7F7FE3C1D7C57FC3D6D7E87FD7D9D6C4E4C3C5C4
7FC2E87FE2C1E27FD9C5D3C5C1E2C57FF94BF1F07F7F7F7F7FD6D57FF1F17AF1F07FE6C5
C4D5C5E2C4C1E86B7FC1E4C7E4E2E37FF1F76B7FF2F0F0F57F7F7F4B00
46
C8000000000000000000C3D6E4D5E3C5D94000000C2C00002B000000000042E20129A03A
48000000881206010000
10
E9000000000000000000
...
PROC PDSCOPY INTAPE INDD=TOPDSCPY OUTDD=SASLIB; RUN;
```

EXECUTING THE TEST

When we execute the test, the first argument to MODULE can be a quoted string starting with a *. If that is used, this indicates that special flags are being passed to MODULE. The flag of *I indicates that debug messages are to appear. These are useful when developing the test. Here is what the debug output looks like for the first iteration on a Linux machine:

```
---PARM LIST FOR MODULEN ROUTINE---
CHR PARM 1 4101D03B 2A69 (*i)
CHR PARM 2 4101D034 636F756E746572 (counter)
---ROUTINE counter LOADED AT ADDRESS 40FC82D4  (PARMLIST AT 4101F00C)---
---VALUES UPON RETURN FROM counter ROUTINE---
---VALUES UPON RETURN FROM MODULEN ROUTINE---
i=1 value=1  (should be equal)
```

But if we want to benchmark the test, we really have to have those hexadecimal pointers such as 4101D03B that were seen earlier. They can change even when run on the same platform. But when we run our tests with the special GENERIC option, those pointers get changed to *GENPTR* as in the following:

```
---PARM LIST FOR MODULEN ROUTINE---
CHR PARM 1 *GENPTR* 2A69 (*i)
CHR PARM 2 *GENPTR* 636F756E746572 (counter)
---ROUTINE counter LOADED AT ADDRESS *GENPTR*  (PARMLIST AT *GENPTR*)---
---VALUES UPON RETURN FROM counter ROUTINE---
---VALUES UPON RETURN FROM MODULEN ROUTINE---
i=1 value=1  (should be equal)
```

However, even this type of output can produce host-dependent text. For example, the word ‘counter’ as seen earlier is in ASCII, and on z/OS it would be in EBCDIC, so the hex characters would be different.

So what is best for a benchmark is to remove the *I option, and produce only what should appear on all hosts if the test runs correctly, as in the following:

```
i=1 value=1  (should be equal)
i=2 value=2  (should be equal)
i=3 value=3  (should be equal)
i=4 value=4  (should be equal)
i=5 value=5  (should be equal)
i=6 value=6  (should be equal)
i=7 value=7  (should be equal)
i=8 value=8  (should be equal)
i=9 value=9  (should be equal)
i=10 value=10  (should be equal)
```

CONCLUSION

Using a front-end test program with companion hex streams for each platform enables us to test a system that is heavily host-dependent, but manage it in a portable way.

ACKNOWLEDGMENTS

I would like to thank Mandy Chambers for her assistance in creating and maintaining the testing harness described herein.

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the author at Rick.Langston@sas.com