Paper OO-01-2013

Using Microsoft® Windows® DLLs within SAS® Programs

Rajesh Lal, Experis, Portage, MI, USA

ABSTRACT

SAS has a wide variety of functions and call routines available. More and more Operating System level functionality has become available as part of SAS language and functions over the versions of SAS. However, there is a wealth of other Operating System functionality that can be accessed from within SAS with some preparation on the part of the SAS programmer.

Much of the Microsoft Windows functionality is stored in easily re-usable system DLL (Dynamic Link Library) files. This paper describes some of the Windows functionality which may not be available directly as part of SAS language and methods of accessing that functionality from within SAS code. Using the methods described here, practically any Windows API should become accessible. User created DLL functionality should also be accessible to SAS programs.

INTRODUCTION

SAS® has a wide variety of functions and call routines available including but not limited to mathematical, logical, date/time, statistical, financial, character string management and file management categories. Over time and the versions of SAS, more and more Operating System level functionality has become available as part of SAS language and functions. However, there is a wealth of other Operating System functionality that may not be available directly as part of SAS language.

Dynamic link libraries (DLLs) are executable files that contain one or more routines written in any of several programming languages. DLLs are a mechanism for storing useful routines that might be needed by many applications. When an application needs a routine that resides in a DLL, it loads the DLL, invokes the routine, and unloads the DLL upon completion.

SAS provides the MODULE family of call routines and functions (including MODULE, MODULEN, MODULEC, MODULEI, MODULEIN, and MODULEIC) to invoke a routine that resides in an external DLL from within SAS. The use of these functions requires creation of a text file that describes the DLL routine to be invoked.

OVERVIEW

The following topics will be covered in the rest of the paper:

- The steps needed for accessing an external DLL routine from within SAS:
    - Creating a text file called SASCBTBL Attribute Table that describes the DLL routine we want to access, including the arguments it expects and the values it returns (if any).
    - Using the FILENAME statement to assign the SASCBTBL file reference to the attribute file created in the step above.
    - Using a call routine or function (MODULE, MODULEN, or MODULEC) to invoke the DLL routine.
- Searching for a system DLL routine on MSDN[1]
- Generating the SASCBTBL file from DLL routine definition
- Examples of using the DLL functions from within SAS
    - Creating an encrypted file output
    - Creating an output in an unknown environment based upon software installed
    - Simultaneous write access by multiple users to the different regions of the same file

ACCESSING AN EXTRENAL DLL ROUTINE

Some important aspects of accessing an external DLL routine from within SAS are as described:

[1] Microsoft Developer Network: http://msdn.microsoft.com

SASCBTBL ATTRIBUTE TABLE

The SASCBTBL attribute table is a text file that contains descriptions of the routines we intend to invoke with the MODULE family of SAS functions. The purpose of the table is to define how the MODULE function should interpret its supplied arguments when building a parameter list to pass to the called DLL routine.

The MODULE family of functions invoke an external function that SAS has no knowledge about. Therefore, we must supply information about the external function's arguments so that the MODULE routine can validate them and convert them, if necessary. For example, suppose we want to invoke a routine that requires an integer as an argument. Because SAS uses floating-point values for all of its numeric arguments, the floating-point value must be converted to an integer before we invoke the external routine.

The attribute table should contain a description for each DLL routine we intend to call (using a ROUTINE statement) and the descriptions of each argument associated with the routine (using ARG statements). The syntax of the SASCBTBL attribute table can be found in Appendix 1. The method used to create the SASCBTBL attribute table from the corresponding routine documentation on MSDN will be discussed in later sections.

SASCBTBL FILE REFERENCE

The MODULE routine looks for the attribute information in an attribute table referred to by the SASCBTBL fileref. The MODULE routines locate the table by opening the file referred to by the SASCBTBL fileref. If we do not define this fileref, the MODULE routines simply call the requested DLL routine without altering the arguments.

Using the MODULE functions without defining an attribute table can cause SAS to crash or force us to reset the computer. We need to use an attribute table for all external functions that we want to invoke.

Once the SASCBTBL file has been created, it should be referenced in the program as follows:

```
filename SASCBTBL ‘c:\DLLs\attribtbl.txt’;
```

MODULE FUNCTIONS

After creating the SASCBTBL attribute table and assigning the file reference, the MODULE family of functions can be used to invoke the external DLL routine, inside a DATA step or PROC IML. For example, to encrypt a file, the following ModuleN function call should be made to invoke 'EncryptFileA' routine, which as the name suggests, encrypts a file. The details of this call will be discussed in later section.

```
rc = ModuleN ( '*e'         /* Control parameter */
      , 'EncryptFileA'      /* Name of the routine to be called */
      , fName);             /* the parameter(s) that the routine requires */
```

There are other similar MODULE functions, which can be use for a specific purpose:

- CALL MODULE is used to call the routines that do not return any value
- MODULEN is used to call the routines that returns a numeric value
- MODULEC is used to call the routines that returns a character value
- CALL MODULEI, MODULEIN and MODULEIC are used to call the routines that require vector and matrix arguments and return no value, numeric value and character value respectively.

SEACHING FOR A WINDOWS DLL ROUTINE ON MSDN

MSDN has a tremendous amount of documentation about the routines available in Windows DLL files. Either a search engine can be used to look for particular functionality desired or MSDN can be browsed by the category of functionality desired. For example, File Management functions are located under Data Access and Storage > Local File System > File Management > File Management Reference at:
http://msdn.microsoft.com/library/windows/desktop/bg125389

Figure 1 shows the screen shot of the MSDN page. On the left side of the figure, File Management functions can be found.

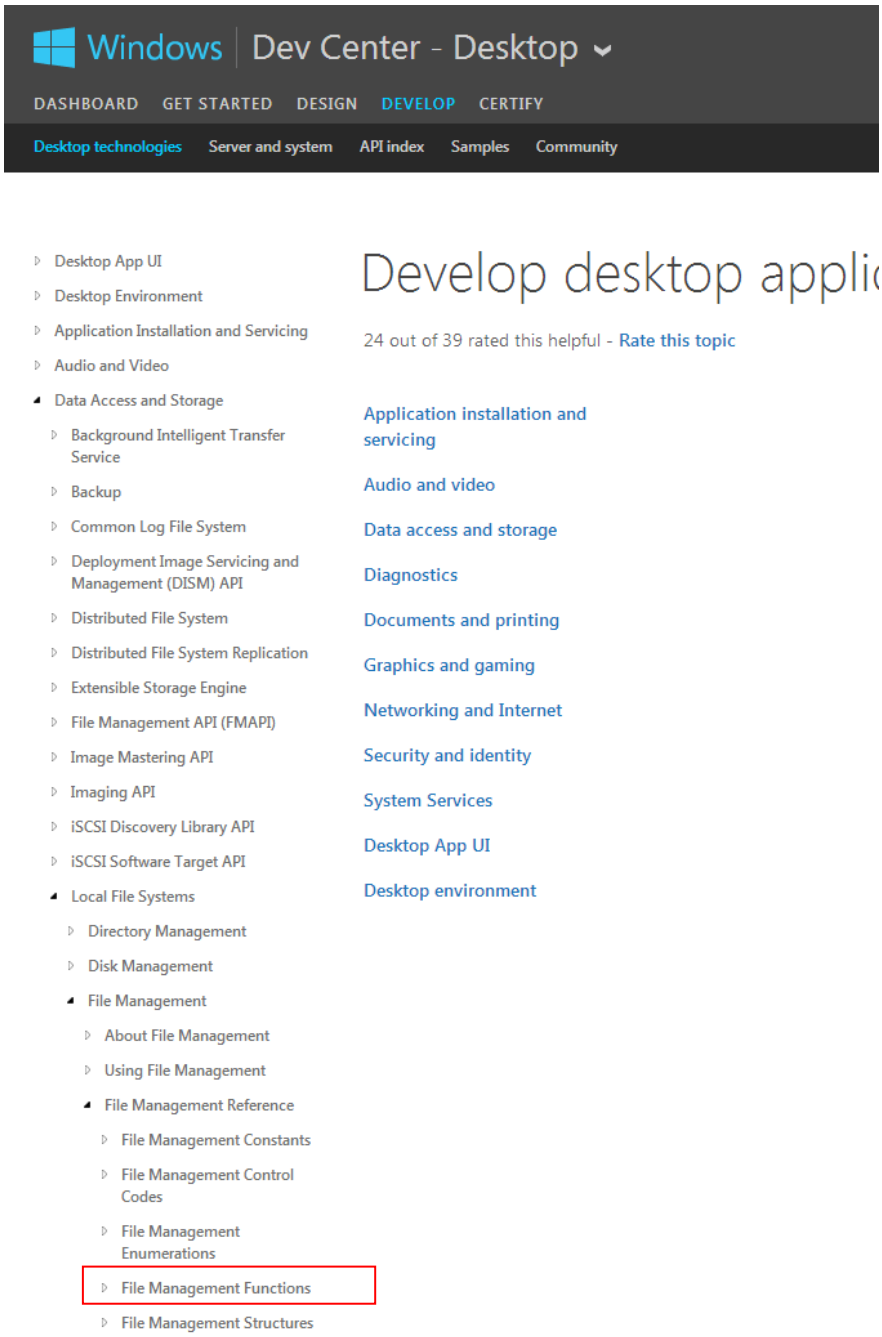

Figure 1. Screen shot of the MSDN page displaying the File Management Functions

GENERATING THE SASCBTBL ATTRIBUTE TABLE FROM DLL ROUTINE DEFINITION

Once the routine of interest has been identified, the following table would help to translate the routine definition into SASCBTBL attribute table:

C Language Data Type	SAS Format/Informat
Double	RB8.
Float	FLOAT4.
signed int	IB4.
signed short	IB2.
signed long	IB4.

char *	IB4. (32 bit SAS)
char *	IB8 (x64 and Itanium SAS)
unsigned int	PIB4.
unsigned short	PIB2.
unsigned long	PIB4.
char[w]	$CHARw. or $CSTRw. (see $CSTRw. Format)

The following is the syntax of RegOpenKeyEx function in Advapi32.dll as defined in MSDN:

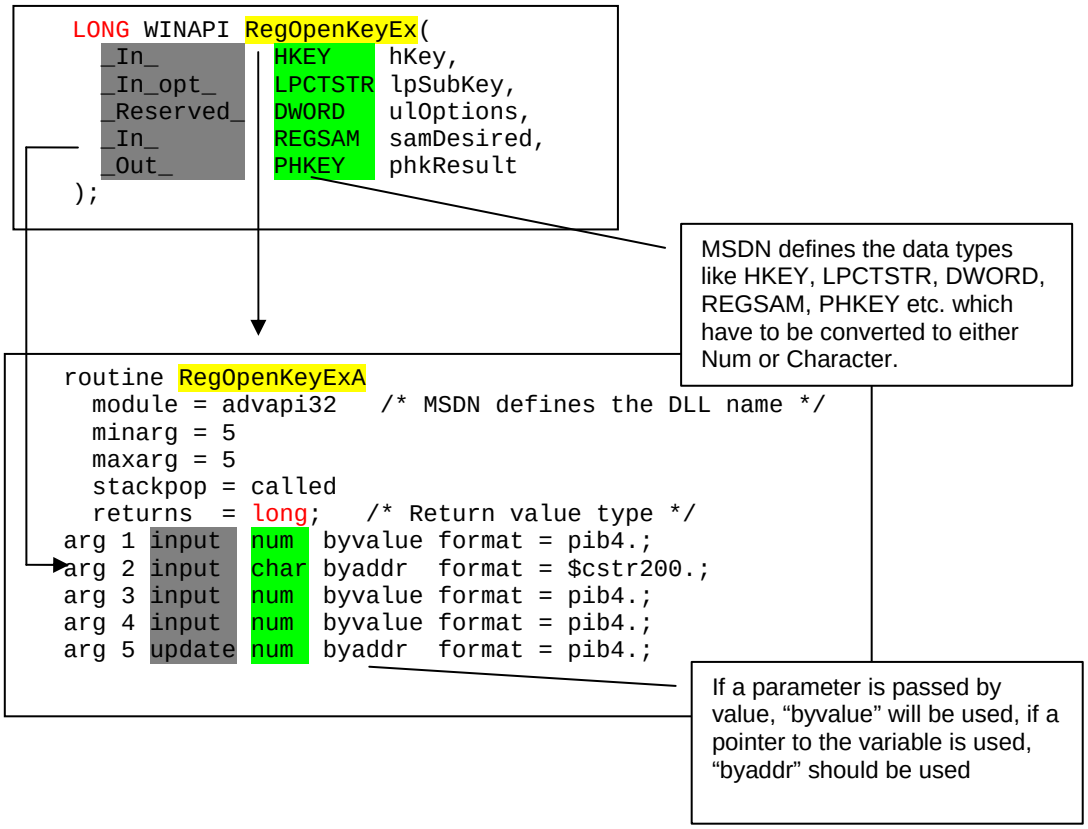

Figure 2. Diagram showing conversion of the routine definition on MSDN to SASCBTBL

- The name of the routine, RegOpenKeyExA, appears in the “routine” statement.
- There are five arguments specified, so minarg and maxarg values will be 5.
- The memory management (stackpop) will usually be “called”.
- The library containing the routine goes in the “module” parameter, which is advapi32.
- Return values is a long integer.
- By comparing the name, order and type of the arguments defined, we define each of the ARG statements.

EXAMPLE 1: CREATING AN ENCRYPTED FILE OUTPUT

SAS provides encryption methods for SAS datasets, but does not directly provide an option for encrypting the output files created from SAS. Encrypting a file is a cryptographic method protection of individual files on NTFS[2] file system volumes using a public-key system. The following routines are available as per MSDN, which can be used for the purpose: Advapi32.dll: EncryptFile, DecryptFile, and AddUsersToEncryptedFile. The following code defines the SASCBTBL attribute table for the EncryptFile & DecryptFile routines and calls the EncryptFile routine to encrypt a file.

[2] New Technology File System is the standard file system of Windows NT, and successor Windows operating systems)

SASCBTBL attribute table:

```
routine EncryptFileA
  module = advapi32
  minarg = 1
  maxarg = 1
  stackpop = called
  returns  = long
;
arg 1 input char format=$cstr200.;   * filename ;

routine DecryptFileA
   module=advapi32
   minarg=2
   maxarg=2
   stackpop=called
   returns=long
;
arg 1 input char format=$cstr200.;
arg 2 input num format=pib4. ;

filename SASCBTBL catalog 'WORK.WINAPI.WINAPI.SOURCE';
```

SAS code calling EncryptFileA routine:

```
DATA _NULL_;
   From_file="C:\Users\rajesh.lal\Desktop\test\textdoc.txt";
   Rc1=modulen('*e','EncryptFileA',From_file);
   Put rc1= ;
Run;
```

The call to EncryptFileA routine encrypts the file using the user's certificate. shows that after encryption of the file, the user who encrypted the file is already added to the list of users who can access the file. AddUsersToEncryptedFile routine can be used to add a new user to the encrypted file. DecryptFileA can be called to decrypt an encrypted file.

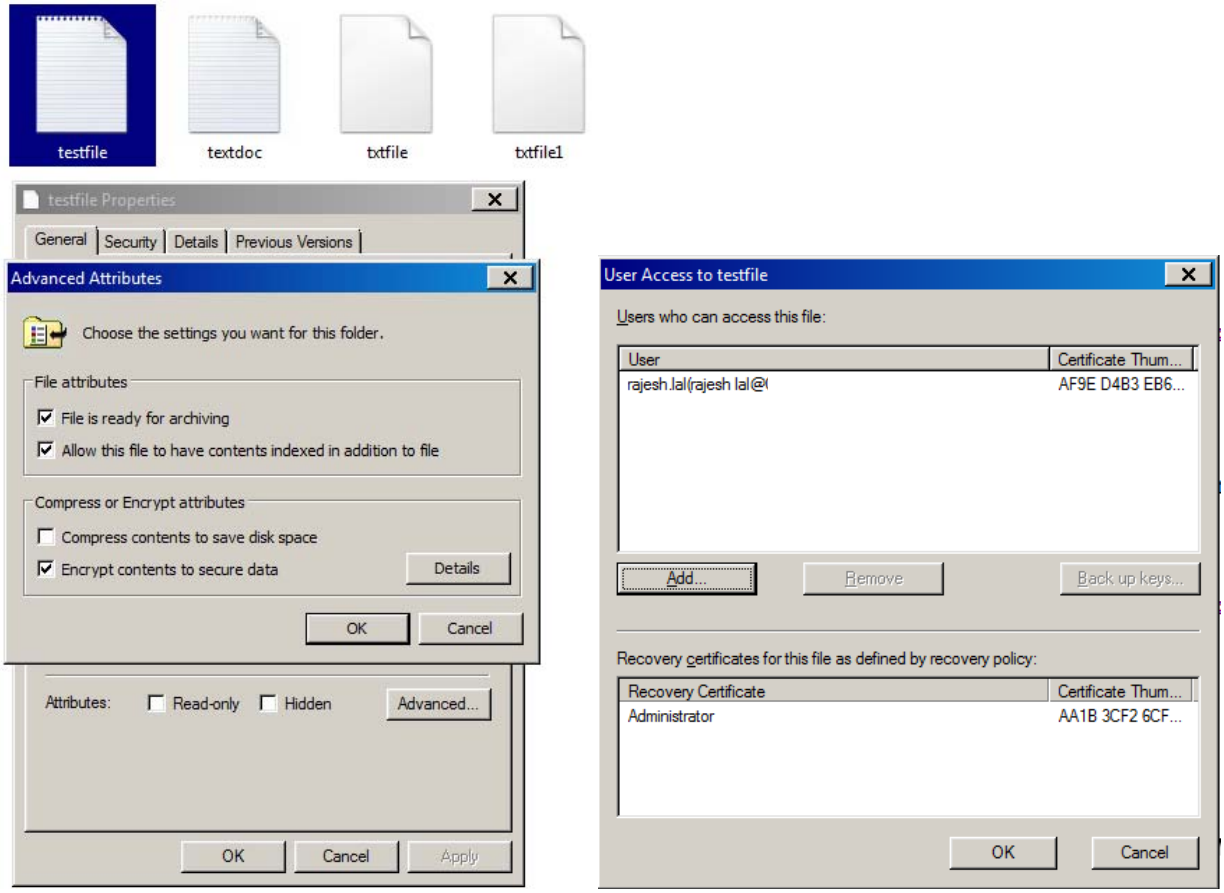

Figure 3. Screen shot of the encrypted file properties

EXAMPLE 2: CREATING AN OUTPUT IN AN UNKNOWN ENVIRONMENT BASED UPON THE SOFTWARE INSTALLED

Consider that a program needs to run in an unknown environment and needs to generate either XLS or XLSX file output (or DOC/DOCX/RTF/TXT), depending upon which version of Microsoft Office® is installed on the system. To solve the problem, we can check the Windows registry location: HKEY_LOCAL_MACHINE\Software\Microsoft\Office\12.0\Common\InstallRoot::Path

Depending upon the value of the registry, we can determine the Microsoft Office version and create the corresponding output file in the SAS program. The following table describes the Office version and the registry key values relationship:

MS Office Version	Registry Key Value
Office 2000	9.0
Office XP	10.0
Office 2003	11.0
Office 2007	12.0
Office 2010	14.0

In Advapi32.dll, there are following routines which can be used to get the registry values: RegOpenKeyExA, RegQueryInfoKeyA. MSDN defines the constant values of HKEY_LOCAL_MACHINE and KEY_QUERY_VALUE.

```
%let HKEY_LOCAL_MACHINE    = 80000002x;
%let KEY_QUERY_VALUE = 00000001x;
node = 'Software\Microsoft\Office\12.0\Common\InstallRoot';
rc = ModuleN ( '*e'
        , 'RegOpenKeyExA'
        , &HKEY_LOCAL_MACHINE
        , node
        , 0
        , &KEY_QUERY_VALUE
        , hkey1 );
```

EXAMPLE 3: SIMULTANEOUS WRITE ACCESS BY MULTIPLE USERS TO DIFFERENT PARTS OF THE SAME FILE

If we have to create a SAS application that provides simultaneous write access by multiple users to different parts of the same file, then most of the file management functions provided by Base SAS would not be usable because Base SAS (without SAS/SHARE) does not support write access to multiple users to different parts of the same file. Windows provides LockFile and UnlockFile functions which support acquiring shared or exclusive access to the specified regions of a file.

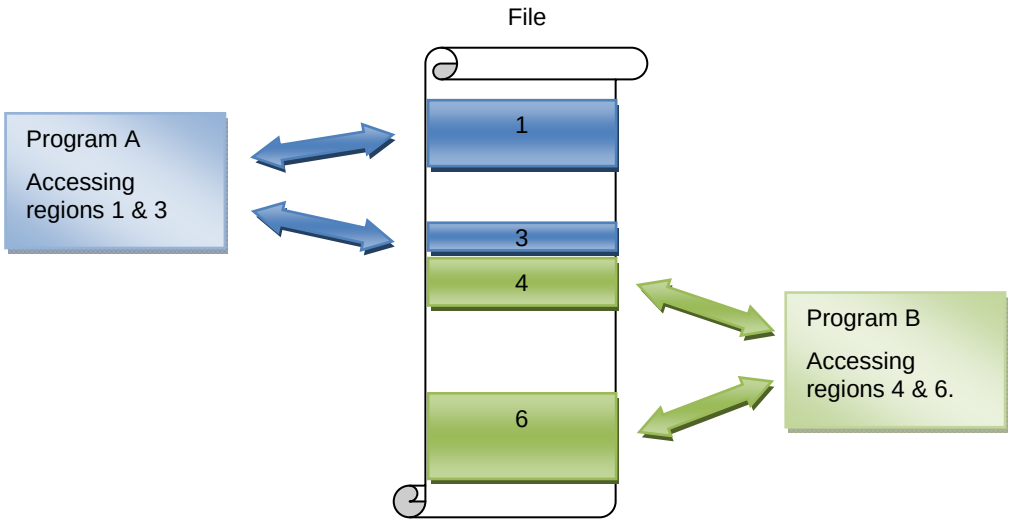

Figure 4. Multiple concurrent programs accessing different regions of the same file

The following code shows the SASCBTBL attribute table definition for CreateFileA, CloseHandle, LockFile and UnlockFile routines and call to each of the routines. Note that the LockFile and UnlockFile routine calls must be made using the same file handle that was created by CreateFileA and also the UnlockFile call should refer to the same file offsets and regions that LockFile referred while locking the file (starting position 10 bytes and ending position 100 bytes).

```
data _null_;
  file SASCBTBL; /* creating the SASCBTBL from within SAS*/
  input ;
  put _infile_;
cards4;
routine CreateFileA
   module=Kernel32
   minarg=7
   maxarg=7
   stackpop=called
   returns=long
;
arg 1 char input format=$cstr256.; *  LPCTSTR lpFileName;
arg 2 num  input format=pib4. byvalue;  * DWORD dwDesiredAccess;
arg 3 num  input format=pib4. byvalue;  * DWORD dwShareMode;
arg 4 num  input format=pib4. byvalue;  * LPSECURITY_ATTRIBUTES
lpSecurityAttributes (set to null,pass 0 byvalue);
arg 5 num  input format=pib4. byvalue;  * DWORD dwCreationDispostion;
arg 6 num  input format=pib4. byvalue;  * DWORD dwFlagsAndAttributes (set to zero);
arg 7 num  input format=pib4. byvalue;  * HANDLE hTemplateFile (ignored);

routine CloseHandle
   module=Kernel32
   minarg=1
   maxarg=1
   stackpop=called
   returns=long
;
arg 1 num  input format=pib4. byvalue;

routine LockFile
   module=Kernel32
   minarg=5
   maxarg=5
   stackpop=called
   returns=long
;
arg 1 num  input format=pib4. byvalue;
arg 2 num  input format=pib4. byvalue;
arg 3 num  input format=pib4. byvalue;
arg 4 num  input format=pib4. byvalue;
arg 5 num  input format=pib4. byvalue;

routine UnlockFile
   module=Kernel32
   minarg=5
   maxarg=5
   stackpop=called
   returns=long
;
arg 1 num  input format=pib4. byvalue;
arg 2 num  input format=pib4. byvalue;
arg 3 num  input format=pib4. byvalue;
arg 4 num  input format=pib4. byvalue;
arg 5 num  input format=pib4. byvalue;
;;;;
run;

DATA _NULL_;
   From_file="C:\Users\rajesh.lal\Desktop\test\txtfile1";
   GENERIC_ACCESS = 10000000x; *** value for GENERIC_ALL access type constant ;
```

```
   FILE_SHARE = 1;
   OPEN_EXISTING = 3;
*** open the file and get the handle;
   hFile = modulen ('CreateFileA',From_file,GENERIC_ACCESS,0,0,OPEN_EXISTING,0,0);
*** lock the region from byte 10 to 100;
   rc1=modulen ('LockFile', hfile, 0,00000002x, 10, 100);
*** unlock the region from byte 10 to 100;
   rc2=modulen ('UnlockFile',hfile,0,0,10,100);
*** close the file;
   rc3=modulen ('CloseHandle', hFile);
   Put hFile= rc1= rc2= rc3=;
run;
```

CONCLUSION

SAS provides means to access the wealth of other Operating System functionality by using the MODULE family of functions. The paper discussed searching for required functionality on MSDN and converting the routine definition to SASCBTBL attribute table. By calling the external routines from a SAS data step, SAS programs or applications can be designed to be more powerful, dynamic and efficient.

REFERENCES

- SAS Online Documentation
- MDSN: http://msdn.microsoft.com/library/windows/desktop/bg125389
- Richard A. DeVenezia's http://www.devenezia.com/downloads/sas/sascbtbl/
- "By Your Command: Executing Windows DLLs from SAS® Enterprise Guide®" by Darryl Putnam, CACI, Inc. Elkridge, MD http://analytics.ncsu.edu/sesug/2010/CC06.Putnam.pdf
- "SAS® with the Windows API" by David H. Johnson, DKV-J Consultancies Ltd, UK & Australia http://www2.sas.com/proceedings/sugi30/248-30.pdf

APPENDIX 1

Following is the syntax of the ROUTINE statement:

```
ROUTINE name MINARG=minarg MAXARG=maxarg
<CALLSEQ=BYVALUE|BYADDR>
<STACKORDER=R2L|L2R>
<STACKPOP=CALLER|CALLED>
<TRANSPOSE=YES|NO> <MODULE=DLL-name>
<RETURNS=SHORT|USHORT|LONG|ULONG |DOUBLE|DBLPTR|CHAR<n>>
<RETURNREGS=DXAX>;
```

"ROUTINE name" starts the ROUTINE statement. A ROUTINE statement is needed for every DLL function we intend to call using the MODULE function. The value for "name" must match the routine name or ordinal you specified as part of the 'module' argument in the MODULE function, where module is the name of the DLL (if not specified by the MODULE attribute) and the routine name or ordinal. The "name" argument is case sensitive, and is required for the ROUTINE statement.

MINARG=minarg specifies the minimum number of arguments to expect for the DLL routine. In most cases, this value will be the same as MAXARG; but some routines do allow a varying number of arguments. This is a required attribute.

MAXARG=maxarg specifies the maximum number of arguments to expect for the DLL routine. This is a required attribute.

CALLSEQ=BYVALUE | BYADDR indicates the calling sequence method used by the DLL routine. Specify BYVALUE for call-by-value and BYADDR for call-by-address. The default value is BYADDR. FORTRAN and COBOL are call-by-address languages; C is usually call-by-value, although a specific routine might be implemented as call-by-address.

The MODULE routine does not require that all arguments use the same calling method; you can identify any exceptions by using the BYVALUE and BYADDR options in the ARG statement, described later in this section.

STACKORDER=R2L | L2R indicates the order of arguments on the stack as expected by the DLL routine. R2L places the arguments on the stack according to C language conventions. The last argument (right-most in the call syntax) is pushed first, the next-to-last argument is pushed next, and so on, so that the first argument is the first item on the stack when the external routine takes over. R2L is the default value. L2R places the arguments on the stack in reverse order, pushing the first argument first, the second argument next, and so on, so that the last argument is the

first item on the stack when the external routine takes over. Pascal uses this calling convention, as do some C routines.

STACKPOP=CALLER | CALLED specifies which routine, the caller routine or the called routine, is responsible for popping the stack (updating the stack pointer) upon return from the routine. The default value is CALLER (the code that calls the routine). Some routines that use Microsofts __stdcall attribute with 32-bit compilers, require the called routine to pop the stack.

TRANSPOSE=YES | NO specifies whether to transpose matrices with both more than one row and more than one column before calling the DLL routine. This attribute applies only to routines called from within PROC IML with MODULEI, MODULEIC, and MODULEIN.

TRANSPOSE=YES is necessary when calling a routine written in a language that does not use row-major order to store matrices. (For example, FORTRAN uses column-major order.) The default value is NO.

MODULE=DLL-name names the executable module (the DLL) in which the routine resides. The MODULE function searches the directories named by the PATH environment variable. If you specify the MODULE attribute here in the ROUTINE statement, then you do not need to include the module name in the module argument to the MODULE function (unless the DLL routine name you are calling is not unique in the attribute table). The MODULE function is described in MODULE Function: Windows. You can have multiple ROUTINE statements that use the same MODULE name. You can also have duplicate ROUTINE names that reside in different DLLs.

RETURNS=SHORT | USHORT | LONG | ULONG | DOUBLE | DBLPTR | CHAR<n> specifies the type of value that the DLL routine returns. This value will be converted as appropriate, depending on whether you use MODULEC (which returns a character) or MODULEN (which returns a number). The possible return value types are: SHORT (short integer), USHORT (unsigned short integer), LONG (long integer), ULONG (unsigned long integer), DOUBLE (double-precision floating point number), DBLPTR (a pointer to a double-precision floating point number.

CHARn pointer to a character string up to n bytes long. The string is expected to be null-terminated and will be blank-padded or truncated as appropriate. If you do not specify n, the MODULE function uses the maximum length of the receiving SAS character variable.

If you do not specify the RETURNS attribute, you should invoke the routine with only the MODULE and MODULEI call routines. You will get unpredictable values if you omit the RETURNS attribute and invoke the routine using the MODULEN/MODULEIN or MODULEC/MODULEIC functions.

The ROUTINE statement must be followed by as many ARG statements as you specified in the MAXARG= option. The ARG statements must appear in the order that the arguments will be specified within the MODULE routines.

The syntax for each ARG statement is

ARG argnum NUM|CHAR <INPUT|OUTPUT|UPDATE> <NOTREQD|REQUIRED> <BYADDR|BYVALUE> <FDSTART> <FORMAT=format>;

ARG argnum defines the argument number. This attribute is required. Define the arguments in ascending order, starting with the first routine argument (ARG 1).

NUM | CHAR defines the argument as numeric or character. This attribute is required. If you specify NUM here but pass the routine a character argument, the argument is converted using the standard numeric informat. If you specify CHAR here but pass the routine a numeric argument, the argument is converted using the BEST12 informat.

INPUT | OUTPUT | UPDATE indicates the argument is either input to the routine, an output argument, or both. If you specify INPUT, the argument is converted and passed to the DLL routine. If you specify OUTPUT, the argument is not converted, but is updated with an outgoing value from the DLL routine. If you specify UPDATE, the argument is converted and passed to the DLL routine and updated with an outgoing value from the routine.

You can specify OUTPUT and UPDATE only with variable arguments (that is, no constants or expressions).

NOTREQD | REQUIRED indicates whether the argument is required. If you specify NOTREQD, then MODULE can omit the argument. If other arguments follow the omitted argument, indicate the omitted argument by including an extra comma as a placeholder. The REQUIRED attribute indicates that the argument is required and cannot be omitted. REQUIRED is the default value.

BYADDR | BYVALUE indicates the argument is passed by reference or by value.

BYADDR is the default value unless CALLSEQ=BYVALUE was specified in the ROUTINE statement, in that case BYVALUE is the default. Specify BYADDR when using a call-by-value routine that also has arguments to be passed by address.

FDSTART indicates that the argument begins a block of values that is grouped into a structure whose pointer is passed as a single argument. Note that all subsequent arguments are treated as part of that structure until the MODULE function encounters another FDSTART argument.

FORMAT=format names the format that presents the argument to the DLL routine. Any SAS Institute-supplied formats, PROC FORMAT-style formats, or SAS/TOOLKIT formats are valid. Note that this format must have a corresponding valid informat if you specified the UPDATE or OUTPUT attribute for the argument.

The FORMAT= attribute is not required, but is recommended, since format specification is the primary purpose of the ARG statements in the attribute table.

Using an incorrect format can produce invalid results or cause a system crash.

ACKNOWLEDGMENTS

I would like to thank Jack Fuller, Scott Davis and Chuck Kincaid for reviewing this paper and providing valuable comments.

CONTACT INFORMATION

Your comments and questions are valued and encouraged. For any questions or sample codes described in this paper, please contact the authors at:

Name: Rajesh Lal
Enterprise: Experis
Address: 5220 Lovers Lane, STE 200
City, State ZIP: Portage, MI 49002
Work Phone: 269-553-5147
Fax: 269-553-5101
E-mail: rajesh.lal@experis.com
Web: www.experis.com