Using CALL MODULE in SAS® on Linux, or *I get by with a little help from my friends*

Richard A. DeVenezia, Independent Consultant, Remsen, New York
Judy Loren, Health Dialog Analytic Solutions, Portland, Maine

ABSTRACT

An experienced SAS coder can tackle almost any situation. However, there are occasions when a dance, not a tackle, is in order. The CALL MODULE routine is a great dance partner.

A prominent healthcare data analysis firm needed to assign Diagnostic Related Group codes (DRGs) to the claims in their data warehouse. Information and instructions on assigning DRG codes is publicly available, and the firm has SAS coders that could implement the metrics. But the most beneficial solution did not involve writing SAS code to assign the DRGs. Instead, a low-cost module (a C-callable shared object) was purchased from a third party provider in the DRG data processing marketplace. The CALL MODULE routine allowed SAS to use the methods in the object like a data step function.

This paper will explain how to call an external module from SAS on the Linux platform, and will share the lessons learned in implementing the selected DRG grouper.

DRG

DRGs (Diagnosis-Related Groups) were developed originally at Yale University in the 1960s for monitoring quality of care and utilization of services in an inpatient setting. In the 1980s, the now CMS (Centers for Medicare and Medicaid Services, formerly the Health Care Finance Administration, HCFA) chose to use them as the basis for paying hospitals for acute inpatient services among the Medicare population. There are about 540 different DRG's, each of which is assigned a specific weight for purposes of determining the amount of payment. The logic for determining which DRG a specific inpatient event should be assigned is publicly available. It uses as input the age and sex of the patient, the primary and all secondary diagnoses associated with the admission, as well as any surgical procedures performed, and the discharge status of the patient.

DRGs have continued to evolve since the original inception. In fact, a significant expansion is in the works for 2008. In addition, variants proposed by private organizations have appeared on the market. Any organization interested in analyzing cost and/or quality of healthcare, as well as any hospital that bills Medicare for patient services, needs to implement one or more of the DRG methodologies.

Software for assigning DRGs to inpatient events is available from a number of sources. Products differ on mode of use (interactive vs batch) and purpose (assigning a DRG for analysis vs determining allowable price based on DRG, region, type of facility). There is a wide range of price and functionality. Health Dialog found a good ratio of value to price at DRGGroupers.com (http://www.drggroupers.com/index.html). This DRG software vendor offered reasonably priced logic, data driver and C-callable shared object products meeting the requirements of Health Dialog. In addition, they were very responsive to questions and inquiries about the workings of the routine, the content of the DRG logic and the annual masks. Their website contains several free downloads that are a boon to the new user of DRGs.

A very successful batch system for assigning DRG's to claims for acute inpatient services was implemented using methods in a shared object by way of the MODULEN function. The third party cost was nominal (1% of the 3M quoted cost). Significant savings in cost and using esoteric features of SAS -- who could ask for anything more ?

C-CALLABLE SHARED OBJECT

Without going into a lot of detail, a C-callable shared object is simply a function written in C that can be called from another program or function. Similar to the way you call a native SAS function, you supply values of all the necessary arguments at the time of invocation. The function returns a value (or an array of values) to the variable named by the programmer. Calling the DRG grouper from C looks like this:

```
retval = mhdrg(
        "f24",         /* which DRG version you want */
        ".",           /* path to masks files */
        "1",           /* discharge status (home) */
        "34",          /* patient age on admission */
        "2",           /* patient sex (1=male, 2=female) */
        "V3000",       /* ICD DX codes (single live newborn) */
        "    ",        /* ICD procedure codes (none) */
        5,             /* length of each ICD DX code */
        4              /* length of each ICD procedure code */
    );
```

What does it take to allow similar functionality in SAS—calling the exact same object from a SAS program instead of a C program?

EXTERNAL LIBRARIES

External libraries are files that contain precompiled code. The methods within a code library are programmed with the understanding they can be loaded and unloaded dynamically by other programs. The entries within a code library conform to operating system standards for dynamic linking. Loadable code files are most often recognized by filename extension:

- Windows: Dynamic Link Libraries (*.dll)
- UNIX: Shareable Libraries (*.so)

CALL MODULE ROUTINE

SAS DATA Step and SAS/AF SCL programs use the ModuleN() and ModuleC() functions to call routines that reside in external libraries ModuleN() is used when the routine returns a numeric value; ModuleC() is used for character results. Each routine you call must be invoked with the proper number of arguments and types (known as attribute information). The attribute information about each routine you want to call must be explicitly described. The ModuleN and ModuleC function use the attribute information to perform internal conversions needed to communicate with the external object according to operating system standards.

SAS locates the explicit descriptions using the fileref SASCBTBL. More details on this file appear later in this paper.

SAS CODE

This paper will present an overview of the SAS features that allow access to externally written routines. They are explained in the context of the specific example of calling the DRG grouper on a Linux system. We will start by showing the SAS code to use the DRG grouper. Each section is annotated with comments, to allow you to connect the statements in this SAS program to the features described and explained below.

```
* Add the path containing the objects required by the DRG function to the Linux path search;
x "setenv LD_LIBRARY_PATH .:/mypath/production/objects";

* This file contains the instructions on how to call the DRG function;
filename SASCBTBL "/mypath/production/objects/cbtbl.txt";

* Start a data step, reading a dataset whose records have the necessary information for the arguments;
data grouped;
      set inpatient_claims;

* Translate character values of sex to numeric digit expected in mhdrg;
      if sex = 'M' then sex_num = '1';
      else if sex = 'F' then sex_num = '2';
      else sex_num = '3';
```

```
* Create an undelimited list of diagnosis codes and a similar list of procedure codes;
      all_dx = cat(of dx1-dx10);
      all_proc = cat(of icd9proc1-icd9proc15);

* Calculate which DRG Mask to use, based on the admit date;
      version = put(adm_dt,drgmask.);

* Call the function that yields the DRG information.;
* Note: The 9th parameter to MODULEC is the length of a given Dx value. Used to parse the Dx list;
* Note: The 10th parameter to MODULEC is the length of a given ICD9Proc value. Used to parse the ICD9Proc list;
      retval = MODULEC ('mhdrg'
       , version, &PATH., dsch_status, age, sex_num, all_dx, all_proc, 5, 4);

* If there is an error, write a message to the log and set the return values to missing;
      if retval = "" then do;
            put "ERROR: Problem invoking mhdrg";
            errmsg = MODULEC('errdesc');
            put "ERROR:" errmsg;
            call missing(rc, drg_mdc, drg, drg_ovn, drg_weight, drg_mean,
            drg_type,  drg_desc);
      end;
      else do;
* Parse the successfully returned value, a string of DRG results delimited by a ^ (caret);
            rc = scan(retval, 1, '^');
            drg_mdc = scan(retval, 2, '^');
            * ensure there are leading zeros;
            drg = put(input(scan(retval, 3, '^'), best3.), z3.);
            drg_ovn = input(scan(retval, 4, '^'), best12.);
            drg_weight = input(scan(retval, 5, '^'), best12.);
            drg_mean = input(scan(retval, 6, '^'), best12.);
            drg_type = scan(retval, 7, '^');
            drg_desc = scan(retval, 8, '^');
      end;
      output; * will have one record per BY group.;
run;
```

TELLING SAS WHERE TO FIND THE SHARED OBJECT

Operating system standards require a shared object be located in a directory that is listed in the operating system's **library path environment variable.** The following table lists this environment variable for each UNIX operating system that SAS supports.

Operating System	**Library path environment variable**
Solaris	LD_LIBRARY_PATH
AIX/R	LIBPATH
HP-UX	LD_LIBRARY_PATH or SHLIB_PATH
Linux	LD_LIBRARY_PATH
Windows (*.dll)	PATH

In this case the SAS program runs under Linux, so the location of the shared object must be added to the library paths listed in environment variable LD_LIBRARY_PATH. This is accomplished with the x command:

```
      x "setenv LD_LIBRARY_PATH .:/mypath/production/objects";
```

TELLING SAS HOW TO COMMUNICATE WITH THE SHARED OBJECT: SASCBTBL FILEREF

SAS requires a detailed description of an external method in order to use it. I don't know what SASCBTBL stands for exactly, I think of it as "**SAS C**ontrol **B**lock **T**a**BL**e." However, according to a birdie:

> It's actually an abbreviation for **SAS C**o**B**ol **T**a**BL**e. Why that? The original version of MODULE was called COBOLINT. It was intended as a COBOL interface for sites that wanted to call COBOL routines from within the DATA step.

Introduction to the SASCBTBL Attribute Table

From the SAS Companion for Unix Environments comes an excellent introduction to the SASCBTBL Attribute Table:

> "Because the MODULE function invokes an external routine that SAS knows nothing about, you must supply information about the routine's arguments so that the MODULE function can validate them and convert them, if necessary. For example, suppose you 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 you invoke the external routine. The MODULE function looks for this attribute information in an attribute table referred to by the SASCBTBL fileref. The attribute table is a sequential text file that contains descriptions of the routines you can invoke with the CALL MODULE function. The MODULE function locates the table by opening the file that is referenced by the SASCBTBL fileref. If you do not define this fileref, the MODULE function simply calls the requested shared library routine without altering the arguments"

Because SASCBTBL is a fileref, the actual location of the text file can be pretty much anywhere -- an operating system file, a SAS catalog SOURCE entry, or a file on an FTP site. In almost every situation an operating system file is an excellent choice.

The attribute table should contain a description for each external routine that you intend to call, and descriptions of each argument associated with that routine.

Some fileref tricks include concatenating files or wildcarding files. In rare cases these features can be used when there is a lot of attribute information that is best managed when kept separated. For the case of several attribute table files in a single directory:

    filename SASCBTBL "/mypath/production/objects/cbtbl-*"; %* wildcarded concatenation;

Or for the case of attribute table files in separate directories:

    filename SASCBTBL ("/mypath/experimental/cbtbl.txt" "/mypath/production/objects/cbtbl.txt"); %*
concatenated;

Be polite -- if you are coding in a large system that already uses SASCBTBL, be sure to restore the fileref to its original setting when you are done using it.

Quoting once again from the SAS documentation:

> **"CAUTION: Using the MODULE function without defining an attribute table can cause SAS to crash, produce unexpected results, or result in severe errors."**

Clearly, this is not for the faint of heart! The clues in this paper are intended to help you avoid such disasters.

Contents of cbtbl.txt for the DRG grouper:

The attribute file must be assigned the fileref sascbtbl:

```
filename sascbtbl "/mypath/production/objects/cbtbl.txt";
```

The cbtbl.txt file for the DRG grouper looks like this:

```
routine mhdrg
minarg = 9
maxarg = 9
module = slib
```

```
returns = CHAR200;
arg 1 char input byaddr format=$cstr10. ;
arg 2 char input byaddr format=$cstr200. ;
arg 3 char input byaddr format=$cstr20. ;
arg 4 char input byaddr format=$cstr3. ;
arg 5 char input byaddr format=$char2. ;
arg 6 char input byaddr format=$char50. ;
arg 7 char input byaddr format=$char60. ;
arg 8 num input byvalue format=ib4.   ;
arg 9 num input byvalue format=ib4.   ;

routine errdesc
minarg = 0
maxarg = 0
module = slib
returns = CHAR200;
```

This table defines two routines: mhdrg (starting on the first line) and errdesc (starting on line 15). Both routines are found in the object file named ***`slib`*** stored in the directory `mypath/production/objects` (which is in the library path setup with the command in the X statement).

Syntax of the Attribute Table

The following sections are excerpted from the SAS Companion to Unix Environments:

“The attribute table should contain the following:

a description in a ROUTINE statement for each shared library routine you intend to call

descriptions in ARG statements for each argument associated with that routine.

At any point in the attribute table file, you can create a comment using an asterisk (*) as the first nonblank character of a line or after the end of a statement (following the semicolon). You must end the comment with a semicolon.”

ROUTINE Statement in CBTBL Table

“The syntax of the ROUTINE statement is

```
ROUTINE name MINARG=minarg MAXARG=maxarg  <CALLSEQ=BYVALUE|BYADDR>
<TRANSPOSE=YES|NO> <MODULE=shared-library-name>
<RETURNS=SHORT|USHORT|LONG|ULONG |DOUBLE|DBLPTR|CHAR<n>>
```

After the keyword ROUTINE, supply the name of the routine in the shared library that you plan to use. In this case, the name of the first routine defined is mhdrg; another routine (errdesc) is defined in the same cbtbl file.

MINARG=minarg specifies the minimum number of arguments to expect for the shared library 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.” *The mhdrg routine requires exactly 9 arguments, so the value 9 is given for both the minarg and the maxarg components.*

“**CALLSEQ=BYVALUE | BYADDR** indicates the calling sequence method used by the shared library routine. Specify BYVALUE for call-by-value and BYADDR for call-by-address. The default value is BYADDR. The MODULE function 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.” *(which is shown in this example; see below).*

<TRANSPOSE=YES|NO> *will not be discussed here. For more information, see the SAS documentation.*

“**MODULE=shared-library-name** names the executable module (the shared library) in which the routine resides. You do not need to specify this attribute if the name of the shared library is the same name as the routine.*” In the case of the DRG grouper at Health Dialog, the shared library that was received from the vendor was saved with the name slib in the directory path `/mypath/production/objects`.*

You can have multiple ROUTINE statements that use the same MODULE name. In this case, both the mhdrg routine and the errdesc routine came packaged in the slib object.

“**RETURNS=SHORT | USHORT | LONG | ULONG | DOUBLE | DBLPTR | CHAR<n>** specifies the type of value that the shared library routine returns. *Since the DRG grouper returns a string of values which we will use SAS to parse, it is defined as a long character variable (char200).* This accepts 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.”

ARG Statement in CBTBL Table

Continuing the excerpt from the SAS Companion to Unix Environments:

“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 function.

The syntax for each ARG statement is

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

Following the keyword ARG, give the number of each argument in the order the routine expects them.

Specify whether the argument is expected as NUMeric or CHARacter. “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.

The next portion of the ARG statement, “**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 shared library routine. If you specify OUTPUT, the argument is not converted, but is updated with an outgoing value from the shared library routine. If you specify UPDATE, the argument is converted, passed to the shared library routine and updated with an outgoing value from the routine.

“**NOTREQD | REQUIRED** indicates whether the argument is required. The REQUIRED attribute indicates that the argument is required and cannot be omitted. REQUIRED is the default value.

“If you specify NOTREQD, then the MODULE function can omit the argument. If other arguments follow the omitted argument, identify the omitted argument by including an extra comma as a placeholder. For example, to omit the second argument to routine XYZ, you would specify:

```
call module('XYZ',1,,3);
```

“**CAUTION: Be careful when using NOTREQD; the shared library routine must not attempt to access the argument if it is not supplied in the call to MODULE. If the routine does attempt to access it, you might receive unexpected results or severe errors.**

“**BYADDR | BYVALUE** indicates whether the argument is passed by reference or by value. BYADDR is the default value unless CALLSEQ=BYVALUE was specified in the ROUTINE statement, in which case BYVALUE is the default.” *It required some experimentation to determine exactly how each argument was implemented for the DRG grouper. Specifying the first 7 arguments as call-by-address and the last 2 as call-by-value allowed the mhdrg routine to be invoked properly.*

“**FORMAT=format** names the format that presents the argument to the shared library routine. Any SAS 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.

“**CAUTION: Using an incorrect format can produce invalid results, cause SAS to crash, or result in serious errors.**”

End of material excerpted from SAS online documentation.

PREPARING THE ARG STATEMENTS

The documentation that comes with the shared object library is the best source of information for preparing the ARG statements. The proper argument options were inferred from this sample code found in documentation:

```
retval = mhdrg
( "f14" /* which DRG version you want */
, "/masks" /* path to masks files */
, "8"  /* discharge status (died) */
, "75" /* patient age on discharge */
, "1"  /* patient sex (1=male, 2=female) */
, "7070 1505 5070 518825990 2859 427891980 1976 1977 " /* ICD DX codes */
, "862299609962896593949396" /* ICD procedure codes */
, 5 /* length of each ICD DX code */
, 4 /* length of each ICD procedure code */
);
```

CALL MODULE

The syntax for the MODULEC function is as follows:

```
MODULEC(<cntl-string,>module-name<,argument-1, ..., argument-n>)
```

cntl-string is an optional control string used for showing debugging information in the log. The first character must be an asterisk (*), followed by any combination of the following characters (`IEH`). **`I`** prints the hexadecimal representations of all arguments, I implies E. **`E`** prints detailed error messages. **`H`** provides brief help information.

module-name is the name of the external module to use. There must be a ROUTINE description in the attribute table that exactly matches module-name.

arguments are character or numeric expressions that correspond to the ARG statements in the attribute table.

During development of the batch system Richard was called upon to explain the log when cntl-string '*IE' was used. In this log entry the ARG format for the last two arguments was IB2. IB2. was chosen on the assumption the arguments were short-ints (2 bytes) as indicated in the mdhrg documentation. The format actually needed is IB4. which corresponds to a 4 byte integer. The discrepancy could be from either outright misinformation in the documentation, or perhaps differences in platforms and compilers used to create the loadable library.

The first part logs the arguments as passed from SAS DATA Step to SAS MODULEC function. This is not the same as the arguments eventually passed to the routine in the .so library. The first one (or two) are used to control what MODULEC does.

```
---PARM LIST FOR MODULEC ROUTINE---
CHR PARM 1 027C95C2 2A6965 (*ie)
CHR PARM 2 027C95BD 6D68647267 (mhdrg)

CHR PARM 3 027C978C 663234
663234 is "f24" at memory address 027C978C

CHR PARM 4 027C978F 2E
2E is "." at memory address 027C978F

CHR PARM 5 027C9790 31
31 is "1" at memory address 027C9790

CHR PARM 6 027C9791 3334
3334 is "34" at memory address 027C9791

CHR PARM 7 027C9793 32
32 is "2"  at memory address 027C9793

CHR PARM 8 027C9794 5633303030
5633303030 is "V3000" at memory address 027C9794
```

```
CHR PARM 9 027C9799 20202020
20202020 is four spaces at memory address 027C9799

NUM PARM 10 027C9484 0000000000001440
0000000000001440 is 5 in ieee-754 double representation at memory address 027C9484

NUM PARM 11 027C947C 0000000000001040
0000000000001440 is 4 in ieee-754 double representation at memory address 027C947C
```

The MODULEC routine takes its incoming arguments and copies them into dynamically allocated memory for 'passing' to the desired module specified in arg1 (or 2)

```
---ROUTINE mhdrg LOADED AT ADDRESS 01F5A1D8  (PARMLIST AT 027BFC0C)---
PARM 1 027B1808
6632340000000000000000000000000000000000000000000000000000000000000000 … and lots more
zeros …
"f24" was copied into 200 bytes of storage at memory address 027B1808
200 comes from $cstr200. in routines arg declaration

PARM 2 027B18D0
2E00000000000000000000000000000000000000000000000000000000000000000000 … and lots more
zeros …
"." was copied into 200 bytes of storage at memory address 027B18D0
200 comes from $cstr200. in routines arg declaration

PARM 3 027B1998
3100000000000000000000000000000000000000000000000000000000000000000000 … and lots more
zeros …
"1" was copied into 200 bytes of storage at memory address 027B1998
200 comes from $cstr200. in routines arg declaration

PARM 4 027B1A60
3334000000000000000000000000000000000000000000000000000000000000000000 … and lots more
zeros …
PARM 5 027B1B28
3200000000000000000000000000000000000000000000000000000000000000000000 … and lots more
zeros …
PARM 6 027B1BF0
5633303030000000000000000000000000000000000000000000000000000000000000 … and lots more
zeros …
PARM 7 027B1CB8
0000000000000000000000000000000000000000000000000000000000000000000000 … and lots more
zeros …

PARM 8 027B1D80 0500
Number 5 in double precision numeric representation was converted to Number 5 in 2
byte binary representation and stored at memory location 027B1D80 (and logged using
HEX representation)

PARM 9 027B1D82 0400
Number 4 in double precision numeric representation was converted to Number 4 in 2
byte binary representation and stored at memory location 027B1D80 (and logged using
HEX representation)
```

CREATING C-CALLABLE SHARED OBJECT

An organization with in-house C coding expertise can create a shared object (or .dll) that is easily used from within a SAS session. On the Linux platform the gcc compiler is freely available and can create shared objects. On the Windows platform there are both free and commercial compilers. The following example .dll project was created using Microsoft Visual C++® 5.0 in Microsoft Developer Studio 97.

The problem: A SAS/AF frame must run a termination section that releases system resources. The termination section does not run when a SAS session is ended by way of the system close menu.

The project: Disable the system close menu in all windows of a SAS session.

Routines in the Microsoft Windows Win32 library can be used to discover the handles of the windows of a SAS session. However, these routines use a callback feature, and thus cannot be utilized directly from within SAS using the ModuleN routine. Callbacks are easy to program in C.

Various project options are used to cause a C compiler to create a .dll instead of an .exe. The easiest way to set these options is to use a wizard or project template. In VC++5 the menu File/New..Projects/Win32 Dynamic-Link Library was used to create a new dll project named "dcb" (disable close button). The output of the project is the file dcb.dll that is later copied to a folder in the system PATH.

The C code:

```
BOOL CALLBACK CallbackForDCB ( HWND hWnd, LPARAM lParam )
{
      // lParam contains the id of the process that wants it's windows
      // system menu changed to disallow the close item

      DWORD pid ;
      HMENU hMenu ;

      // Determine the id of the process that owns the window
      GetWindowThreadProcessId ( hWnd, &pid );

      // Determine if the process is the process of interest
      if (pid == (DWORD) lParam)
      {
            // Obtain a handle to the windows system menu
            hMenu = GetSystemMenu(hWnd, FALSE) ;
            if (hMenu != NULL) {
                  // Grey the close item (will also grey the X icon)
                  EnableMenuItem (hMenu, SC_CLOSE, MF_BYCOMMAND | MF_GRAYED);
            }
      }
      return TRUE;  // returning true indicates the enumeration should continue
}

void DisableCloseButton() // DCB
{
      DWORD pid;

      // Get the process id of the calling process
      pid = GetCurrentProcessId ();

      // Invoke the top level window enumerator.
      // EnumWindows will invoke the callback procedure for each top level window,
      // passing in the HWND and the pid
      EnumWindows ( CallbackForDCB, pid );
}
```

The SASCBTBL:

```
routine DisableCloseButton
  module = dcb
  minarg = 0
  maxarg = 0
  stackpop = called;
```

The routine statement does not use the RETURNS= attribute. Why? Because the method is of type void.

The SAS code:

```
data _null_;
  call module ('DisableCloseButton');
run;
```

According to documentation "If you do not specify the RETURNS attribute, you should invoke the routine with only the MODULE and MODULEI call routines." The external method <u>can</u> be invoked with the MODULEN function, with the understanding that the returned value can be unpredictable and should not be used.

CONCLUSION

The CALL MODULE routine is a powerful SAS feature for taking advantage of methods written in other languages but designed to be accessed dynamically. Health Dialog found a code library in the DRG data marketplace that saved them thousands of dollars and used it to compute necessary metrics. Using the same principles, then, any custom-written code library is readily accessible for use in your SAS programs.

You too can get by with a little help from your friends.

REFERENCES

Federal Register, Friday, August 30, 2002,
http://www.cms.hhs.gov/LongTermCareHospitalPPS/downloads/55954-56002.pdf

SAS Documentation, "Accessing External Shareable Images from SAS : The SASCBTBL Attribute Table"

SAS Companion for Unix Environments, Online Documentation, Accessing Shared Executable Libraries from SAS, The SASCBTBL Attribute Table,
http://support.sas.com/onlinedoc/913/docMainpage.jsp

MSDN Library
http://msdn.microsoft.com/library/default.asp

CONTACT INFORMATION

Richard A. DeVenezia
9949 East Steuben Road
Remsen, NY 13438
(315) 831-8802
http://www.devenezia.com/contact.php

Richard is an independent consultant who has worked extensively with SAS products for over fifteen years. He has presented at previous SUGI, NESUG and SESUG conferences. Richard is interested in learning and applying new technologies. He is a SAS-L Hall of Famer and remains an active contributor to SAS-L.

Judy Loren
Health Dialog Analytic Solutions
2 Monument Square
Portland, ME 04101

Judy is a Senior Research Analyst, currently working with a team to develop a system for evaluating the effectiveness and efficiency of individual providers. She has presented at, and chaired sections of, previous SUGI, NESUG, SESUG, and SCSUG conferences.

Disclaimer: The authors have no affiliation with or financial interest in DRGGroupers.