



CHAPTER

10

Accessing External DLLs from the SAS System

<i>Overview of Dynamic Link Libraries in the SAS System</i>	165
<i>The SASCBTBL Attribute Table</i>	166
<i>Syntax of the Attribute Table</i>	166
<i>ROUTINE Statement</i>	166
<i>ARG Statement</i>	169
<i>The Importance of the Attribute Table</i>	170
<i>Special Considerations When Using External DLLs</i>	171
<i>Using PEEK Functions to Access Character String Arguments</i>	171
<i>Accessing External DLLs Efficiently</i>	172
<i>Grouping SAS Variables as Structure Arguments</i>	173
<i>Using Constants and Expressions as Arguments to MODULE</i>	174
<i>Specifying Formats and Informats to Use with MODULE Arguments</i>	175
<i>C Language Formats</i>	175
<i>FORTTRAN Language Formats</i>	175
<i>PL/I Language Formats</i>	176
<i>COBOL Language Formats</i>	176
<i>\$CSTRw. Format</i>	177
<i>\$BYVALw. Format</i>	177
<i>Understanding MODULE Log Messages</i>	178
<i>Examples</i>	180
<i>Updating a Character String Argument</i>	180
<i>Passing Arguments by Value</i>	181
<i>Using PEEKC to Access a Returned Pointer</i>	182
<i>Using Structures</i>	182
<i>Invoking a DLL Routine from PROC IML</i>	183

Overview of Dynamic Link Libraries in the SAS System

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. The SAS System under OS/2 provides routines and functions that let you invoke these external routines from within the SAS System. You can access the DLL routines from the DATA step, the IML procedure, and SCL code. You use the MODULE family of SAS call routines and functions (including MODULE, MODULEN, MODULEC, MODULEI, MODULEIN, and MODULEIC) to invoke a routine that resides in an external DLL. This section refers to the MODULE family of call routines and functions generically as the MODULE_{xy} functions.

The general steps for accessing an external DLL routine are

- 1 Create a text file that describes the DLL routine that you want to access, including the arguments that it expects and the values that it returns (if any). This attribute file must be in a special format, as described in “The SASCBTBL Attribute Table” on page 166.
- 2 Use the FILENAME statement to assign the SASCBTBL fileref to the attribute file that you created.
- 3 In a DATA step or in SCL code, use a call routine or function (MODULE, MODULEN, or MODULEC) to invoke the DLL routine. The specific function that you use depends on the type of expected return value (none, numeric, or character). (You can also use MODULEI, MODULEIN, or MODULEIC within a PROC IML step.) The MODULE xy functions are described in “MODULE xy ” on page 255.

CAUTION:

Only experienced programmers should access external DLLs. By accessing a function in an external DLL, you transfer processing control to the external function. If this is done improperly, or if the external function is not reliable, you might lose data or have to reset your computer (or both). Δ

The SASCBTBL Attribute Table

Because the MODULE xy routine invokes an external function that works independently of the SAS System, you must supply information about the function’s arguments so that the MODULE xy routine 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 the SAS System 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 xy routine looks for this attribute information in an attribute table that the SASCBTBL fileref points to.

The attribute table is a sequential text file that contains descriptions of the routines that you can invoke with the MODULE xy function. The function of the table is to define how the MODULE xy function should interpret its supplied arguments when it is building a parameter list to pass to the called DLL routine.

The MODULE xy routines locate the table by opening the file that the SASCBTBL fileref points to. If you do not define this fileref, the MODULE xy routines simply call the requested DLL routine without altering the arguments.

CAUTION:

Using the MODULE xy functions without defining an attribute table can cause the SAS System to fail or force you to reset your computer. You need to use an attribute table for all external functions that you want to invoke. Δ

The attribute table should contain a description for each DLL routine that you intend to call (using a ROUTINE statement) plus descriptions of each argument that is associated with that routine (using ARG statements).

Syntax of the Attribute Table

At any point in the attribute table file, you can create a comment by 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

The syntax of the ROUTINE statement is

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

The following are descriptions of the ROUTINE statement attributes:

ROUTINE *name*

starts the ROUTINE statement. You need a ROUTINE statement for every DLL function that you intend to call using the MODULE.*xy* function. The value for *name* must match the routine name or ordinal that you specified as part of the 'module' argument in the MODULE.*xy* function, where *module* is the name of the DLL (if it was not specified by the MODULE attribute, it must be described later) and the routine name or ordinal. For example, to be able to specify **KERNEL32,GetPath** in the MODULE.*xy* function call, the ROUTINE *name* should be **GetPath**.

The *name* argument is case sensitive, and it 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 is 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 that is 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 executes. 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 executes. 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 (that is, the code that calls the routine). Some routines, such as those that use the Microsoft `__stdcall` attribute with 32-bit compilers, require the called routine to pop the stack.

TRANSPPOSE=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 that are called from within PROC IML with MODULEI, MODULEIC, and MODULEIN.

TRANSPPOSE=YES is necessary when SAS is calling a routine that is written in a language that does not use row-major order to store matrices. (For example, FORTRAN uses column-major order.)

For example, consider this matrix that has three columns and two rows:

		columns		
		1	2	3
rows	1	10	11	12
	2	13	14	15

PROC IML stores this matrix in memory sequentially as 10, 11, 12, 13, 14, 15. However, FORTRAN routines expect this matrix to be 10, 13, 11, 14, 12, 15.

The default value is NO.

MODULE=*DLL-name*

names the executable module (the DLL) in which the routine resides. The MODULE xy function searches the directories that are 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 that you are calling is not unique in the attribute table). The MODULE function is described in “MODULE xy ” on page 255.

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 (instead of using a floating point register). Consult the documentation for your DLL routine to determine how it handles double-precision floating point values.

CHAR n

pointer to a character string up to n bytes long. The string is expected to be null-terminated, and it will be blank-padded or truncated as appropriate. If

you do not specify n , the `MODULExy` 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 get unpredictable values if you omit the `RETURNS` attribute and invoke the routine by 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 are specified within the `MODULExy` routines.

ARG Statement

The syntax for each `ARG` statement is

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

Here are the descriptions of the `ARG` statement attributes:

`ARG` *argnum*

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

NUM | CHAR

defines the argument as numeric or character. This is a required attribute.

If you specify `NUM` here but you pass the routine a character argument, the argument is converted by using the standard numeric informat. If you specify `CHAR` here but you pass the routine a numeric argument, the argument is converted by using the `BEST12` informat.

INPUT | OUTPUT | UPDATE

indicates the argument is either input to the routine or to 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 it is updated with an outgoing value from the DLL routine. If you specify `UPDATE`, the argument is not only converted and passed to the DLL routine but also updated with an outgoing value from the routine.

You can specify `OUTPUT` and `UPDATE` only with variable arguments (that is, you cannot specify constants or expressions).

NOTREQD | REQUIRED

indicates if the argument is required. If you specify `NOTREQD`, then `MODULExy` can omit the argument. If other arguments follow the omitted argument, indicate the omitted argument by including an extra comma as a place holder. For example, to omit the second argument to routine `XYZ`, you would specify:

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

CAUTION:

Be careful when you use `NOTREQD`; the DLL routine must not attempt to access the argument if it is not supplied in the call to `MODULExy`. If the routine does attempt to access it, your system is likely to fail. Δ

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 which case `BYVALUE` is the default. Specify `BYADDR`

when you are 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 `MODULExy` function encounters another `FDSTART` argument.

FORMAT=*format*

names the format that presents the argument to the DLL routine. Any formats that are supplied by SAS Institute, 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 or can cause a system failure.

\triangle

The Importance of the Attribute Table

`MODULExy` routines rely heavily on the accuracy of the information in the attribute table. If this information is incorrect, unpredictable results can occur (including a system failure).

Consider an example routine `xyz` that expects two arguments: an integer and a pointer. The integer is a code that indicates what action takes place. For example, action 1 means that a 20-byte character string is written into the area that is pointed to by the second argument, the pointer.

Now suppose that you call `xyz` by using the `MODULE` routine but that you also indicate in the attribute table that the receiving character argument is only 10 characters long:

```
routine xyz minarg=2 maxarg=2;
arg 1 input num byvalue format=ib4.;
arg 2 output char format=$char10.;
```

Regardless of the value that was given by the `LENGTH` statement for the second argument to `MODULE`, `MODULE` passes a pointer to a 10-byte area to the `xyz` routine. If `xyz` writes 20 bytes at that location, the 10 bytes of memory that follow the string that was provided by `MODULE` are overwritten, causing unpredictable results:

```
data _null_;
  length x $20;
  call module('xyz',1,x);
run;
```

The call executes, depending on which 10 bytes were overwritten. However, this might also cause you to lose data or cause your system to fail.

Also, note that the `PEEK` and `PEEKC` functions rely on the validity of the pointers you supply. If the pointers are invalid, it is possible that the SAS System could fail. For example, this code would cause a failure:

```
data _null_;
  length c $10;
  /* trying to copy from address 0!!!*/
```

```

c = peekc(0,10);
run;

```

Be sure that your pointers are valid and that they are 32-bit pointers when you use PEEK and PEEKC.

Special Considerations When Using External DLLs

Using PEEK Functions to Access Character String Arguments

Because the SAS language does not provide pointers as data types, you must use the PIB4. format or informat to represent pointers. You can then use the SAS PEEK functions to access the data that are stored at these address values.

For example, suppose that you have a routine that is named `GetPath` in a library that is named `SERVICES.DLL`. It has two arguments, an integer function code and a pointer to a pointer. The function code determines what action `GetPath` will take, and the second argument points to a pointer that will be updated by `GetPath` to refer to a system character string. The calling code in C might be

```

GetPath(1,&stgptr);
printf("GetPath indicates string is
      '%s'.\n",stgptr);

```

Using `MODULE`, the corresponding attribute table entry would be

```

ROUTINE GetPath MINARG=2 MAXARG=2
MODULE=SERVICES;
ARG 1 NUM INPUT BYVALUE FORMAT=PIB4.;
ARG 2 NUM OUTPUT BYADDR FORMAT=PIB4.;

```

The attribute table could be invoked as follows:

```

call module('SERVICES,GetPath',1,stgptr);
put stgptr= stgptr=hex8.;

```

If the pointer value in `STGPTR` is `0035F780`, `STGPTR` would actually be set to the decimal value `3536768`, which is the decimal equivalent of `0035F780`. So the preceding `PUT` statement would produce

```

STGPTR=3536768 STGPTR=0035F780

```

However, you want the data at address `0035F780`, not the value of the pointer itself. To access those data, you need to use the `PEEKC` function.

The `PEEKC` function is given two arguments, a pointer via a numeric variable (such as `STGPTR` above) and a length in bytes (characters). `PEEKC` returns a character string of the specified length that contains the characters at the pointer location.

In the example, suppose that `GetPath` sets the second argument's pointer value to the address of the null-terminated character string `C:\XYZ`. You can access the character data with the following statements:

```

call module('SERVICES,GetPath',1,stgptr);
length path $64;
path = peekc(stgptr,64);
i = index(path,'00'x);
if i then substr(path,i)=' ';
/* path now contains the string */

```

The PEEKC function copies 64 bytes that start at the location that is referred to by the pointer in STGPTR. Because you need only the data up to the null terminator (but not including it), you search for the null terminator with the INDEX function, then assign a blank character including and after that point.

You can also use the \$CSTR format in this scenario to simplify your code slightly:

```
call module('SERVICES,GetPath',1,stgptr);
length path $64;
path = put(peekc(stgptr,64),$cstr64.);
```

The \$CSTR format accepts as input a character string of a specified width. It looks for a null terminator and pads the output string with blanks from that point. For more information, see “\$CSTR w . Format” on page 177.

Accessing External DLLs Efficiently

The MODULE xy routine reads the attribute table that is referenced by the SASCBTBL fileref once per step (DATA step, PROC IML step, or SCL step). MODULE xy parses the table and stores the attribute information for future use during the step. When you use a MODULE xy function, the SAS System searches the stored attribute information for the matching routine and module names. The first time that you access a DLL during a step, the SAS System loads the DLL and determines the address of the requested routine. Each DLL that you invoke stays loaded for the duration of the step, and is not reloaded in subsequent calls. All modules and routines are unloaded at the end of the step. For example, suppose the attribute table had the basic form

```
* routines XYZ and BBB in FIRST.DLL;
ROUTINE XYZ MINARG=1 MAXARG=1 MODULE=FIRST;
ARG 1 NUM INPUT;
ROUTINE BBB MINARG=1 MAXARG=1 MODULE=FIRST;
ARG 1 NUM INPUT;
* routines ABC and DDD in SECOND.DLL;
ROUTINE ABC MINARG=1 MAXARG=1 MODULE=SECOND;
ARG 1 NUM INPUT;
ROUTINE DDD MINARG=1 MAXARG=1 MODULE=SECOND;
ARG 1 NUM INPUT;
```

Then suppose that the DATA step looked like this:

```
filename sascbtbl 'myattr.tbl';
data _null_;
  do i=1 to 50;
    /* FIRST.DLL is loaded only once */
    value = moduln('XYZ',i);
    /* SECOND.DLL is loaded only once */
    value2 = moduln('ABC',value);
    put i= value= value2=;
  end;
run;
```

In this example, MODULN parses the attribute table during DATA step compilation. In the first loop iteration ($i=1$), FIRST.DLL is loaded and the XYZ routine is accessed when MODULN calls for it. Next, SECOND.DLL is loaded and the ABC routine is accessed. For subsequent loop iterations (starting when $i=2$), FIRST.DLL and SECOND.DLL remain loaded, so the MODULN function simply accesses the XYZ and ABC routines. The SAS System unloads both DLLs at the end of the DATA step.

Note that the attribute table can contain any number of descriptions for routines that are not accessed for a given step. This does not cause any additional overhead (apart from a few bytes of internal memory to hold the attribute descriptions). In the previous example, BBB and DDD are in the attribute table but are not accessed by the DATA step.

Grouping SAS Variables as Structure Arguments

A common need when you call external routines is to pass a pointer to a structure. Some parts of the structure might be used as input to the routine, while other parts might be replaced or filled in by the routine. Even though the SAS System does not have structures in its language, you can indicate to MODULExy that you want a particular set of arguments to be grouped into a single structure. You indicate this by using the FDSTART option of the ARG statement to flag the argument that begins the structure in the attribute table. The SAS System gathers that argument and all the arguments that follow (until it encounters another FDSTART option) into a single contiguous block, and passes a pointer to the block as an argument to the DLL routine.

For example, consider the GetClientRect routine, which is part of the Win32 API in USER32.DLL. This routine retrieves the coordinates of a window's client area. This also requires the use of another routine, GetActiveWindow, to get the window handle for the window that you want the coordinates from.

The C prototypes for these routines are

```
HWND GetActiveWindow(VOID);
BOOL GetClientRect(HWND hWnd, LPRECT lprc);
```

In C, the code to invoke them is

```
typedef struct tagRECT {
    int left;
    int top;
    int right;
    int bottom;
} RECT;
/* RECT is a structure variable */
/* more SAS statements          */
/* Need the window handle first */
hWnd=GetActiveWindow();
/* Function call, passing the address */
/* of RECT                          */
GetClientRect(hWnd, &RECT);
```

To call these routines using MODULE, you use the following attribute table entries:

```
routine GetActiveWindow
    minarg=0
    maxarg=0
    stackpop=called
    module=USER32
    returns=ushort;
routine GetClientRect
    minarg=5
    maxarg=5
    stackpop=called
    module=USER32;
arg 1 num input byvalue format=pib4.;
```

```

arg 2 num update fdstart format=ib4.;
arg 3 num update          format=ib4.;
arg 4 num update          format=ib4.;
arg 5 num update          format=ib4.;

```

Then use the following DATA step:

```

filename sascbtbl 'sascbtbl.dat';
data _null_;
  hwnd=modulen('GetActiveWindow');
  call module('GetClientRect',hwnd,left,
             top,right,bottom);
  put left= top= right= bottom=;
run;

```

The use of the FDSTART option in the ARG statement for argument 2 indicates that argument 2 and all subsequent arguments are to be gathered together into a single parameter block.

The output in the log from the PUT statement would be as follows:

```
LEFT=2 TOP=2 RIGHT=400 BOTTOM=587
```

Using Constants and Expressions as Arguments to MODULE

You can pass any kind of expression as an argument to the MODULE_{xy} functions. The attribute table indicates whether the argument is for input, output, or update access.

You can specify input arguments as constants and arithmetic expressions. However, because output and update arguments must be able to be modified and returned, you can pass only a variable for these parameters. If you specify a constant or expression where a value that can be updated is expected, the SAS System issues a warning message pointing out the error. Processing continues, but the MODULE_{xy} routine cannot update a constant or expression argument (this means that the value of the argument that you wanted to update will be lost).

Consider these examples. Here is the attribute table:

```

* attribute table entry for ABC;
routine abc minarg=2 maxarg=2;
arg 1 input format=ib4.;
arg 2 output format=ib4.;

```

Here is the DATA step with the MODULE calls:

```

data _null_;
  x=5;
  /* passing a variable as the      */
  /* second argument - OK          */
  call module('abc',1,x);
  /* passing a constant as the     */
  /* second argument - INVALID     */
  call module('abc',1,2);
  /* passing an expression as the  */
  /* second argument - INVALID     */
  call module('abc',1,x+1);
run;

```

In the preceding example, the first call to MODULE is correct because the variable **x** is updated with the value that the **abc** routine returns for the second argument. The

second call to MODULE is not correct because a constant is passed. MODULE issues a warning that indicates that you have passed a constant, and MODULE passes a temporary location instead. The third call to MODULE is not correct because an arithmetic expression is passed, and this causes a temporary location from the DATA step to be used. The returned value is lost.

Specifying Formats and Informats to Use with MODULE Arguments

You specify the SAS format and informat for each DLL name that is specified in the ROUTINE statement by specifying the FORMAT attribute within the ARG statement. The format indicates how numeric and character values should be passed to the DLL routine and how they should be read back upon completion of the routine.

Usually, the format that you use corresponds to a variable type for a given programming language. The following sections describe the proper formats that correspond to different variable types in various programming languages.

C Language Formats

C Type	SAS Format/Informat
double	RB8.
float	FLOAT4.
signed int	IB4.
signed short	IB2.
signed long	IB4.
char *	IB4.
unsigned int	PIB4.
unsigned short	PIB2.
unsigned long	PIB4.
char[w]	SCHARw. or SCSTRw. (see “SCSTRw. Format” on page 177)

Note: For information about passing character data other than as pointers to character strings, see “SBYVALw. Format” on page 177. Δ

FORTRAN Language Formats

FORTRAN Type	SAS Format/Informat
integer*2	IB2.
integer*4	IB4.
real*4	RB4.

FORTTRAN Type	SAS Format/Informat
real*8	RB8.
character*w	\$CHAR w .

The MODULE routines can support FORTRAN character arguments only if they are not expected to be passed by a descriptor.

PL/I Language Formats

PL/I Type	SAS Format/Informat
FIXED BIN(15)	IB2.
FIXED BIN(31)	IB4.
FLOAT BIN(21)	RB4.
FLOAT BIN(31)	RB8.
CHARACTER(w)	\$CHAR w .

The PL/I descriptions are added here for completeness; that they are listed here does not guarantee that you will be able to invoke PL/I routines.

COBOL Language Formats

COBOL Format	SAS Format/Informat	Description
PIC Sxxxx BINARY	IB w .	integer binary
COMP-2	RB8.	double-precision floating point
COMP-1	RB4.	single-precision floating point
PIC xxxx or Sxxxx	F w .	printable numeric
PIC yyyy	\$CHAR w .	character

The following COBOL specifications might not properly match with the formats that are supplied by the Institute because zoned and packed decimal formats are not truly defined for systems that are based on Intel architecture.

COBOL Format	SAS Format/Informat	Description
PIC Sxxxx DISPLAY	ZD w .	zoned decimal
PIC Sxxxx PACKED-DECIMAL	PD w .	packed decimal

The following COBOL specifications do not have true native equivalents and are usable only in conjunction with the corresponding S370Fxx informat and format. The

specifications allow IBM mainframe-style representations to be read and written in the PC environment.

COBOL Format	SAS Format/Informat	Description
PIC <i>xxxx</i> DISPLAY	S370FZDU <i>w</i> .	zoned decimal unsigned
PIC <i>Sxxxx</i> DISPLAY SIGN LEADING	S370FZDL <i>w</i> .	zoned decimal leading sign
PIC <i>Sxxxx</i> DISPLAY SIGN LEADING SEPARATE	S370FZDS <i>w</i> .	zoned decimal leading sign separate
PIC <i>Sxxxx</i> DISPLAY SIGN TRAILING SEPARATE	S370FZDT <i>w</i> .	zoned decimal trailing sign separate
PIC <i>xxxx</i> BINARY	S370FIBU <i>w</i> .	integer binary unsigned
PIC <i>xxxx</i> PACKED-DECIMAL	S370FPDU <i>w</i> .	packed decimal unsigned

\$CSTR*w*. Format

If you pass a character argument as a null-terminated string, use the \$CSTR*w*. format. This format looks for the last nonblank character of your character argument and passes a copy of the string with a null terminator after the last nonblank character. For example, consider the following attribute table entry:

```
* attribute table entry;
routine abc minarg=1 maxarg=1;
arg 1 input char format=$cstr10.;
```

You can then use the following DATA step:

```
data _null_;
  rc = module('abc', 'my string');
run;
```

The \$CSTR format adds a null terminator to the character string **my string** before passing it to the **abc** routine. This action produces an equivalent to the following attribute entry:

```
* attribute table entry;
routine abc minarg=1 maxarg=1;
arg 1 input char format=$char10.;
```

You can then use the following DATA step:

```
data _null_;
  rc = module('abc', 'my string' || '00'x);
run;
```

The first example is easier to understand and easier to use when you use variable or expression arguments.

The \$CSTR informat converts a null-terminated string into a blank-padded string of the specified length. If the DLL routine is supposed to update a character argument, use the \$CSTR informat in the argument attribute.

\$BYVAL*w*. Format

When you use a MODULE*xy* function to pass a single character by value, the argument is automatically promoted to an integer. If you want to use a character expression in the MODULE*xy* call, you must use the special format or informat called

`$BYVAL w` . The `$BYVAL w` format or informat expects a single character and produces a numeric value, the size of which depends on w . `$BYVAL2.` produces a short, `$BYVAL4.` produces a long, and `$BYVAL8.` produces a double. Consider this example, uses the C language:

```
long xyz(a,b)
  long a; double b;
  {
  static char c = 'Y';
  if (a == 'X')
    return(1);
  else if (b == c)
    return(2);
  else return(3);
  }
```

In this example, the `xyz` routine expects two arguments, a long and a double. If the long is an `x`, the actual value of the long is 88 in decimal. This is because an ASCII `x` is stored as hex 58, and this is promoted to a long, represented as 0x00000058 (or 88 decimal). If the value of `a` is `x`, or 88, a 1 is returned. If the second argument, a double, is `Y` (which is interpreted as 89), then 2 is returned.

Now suppose that you want to pass characters as the arguments to `xyz`. In C, you would invoke them as follows:

```
x = xyz('X', (double)'Z');
y = xyz('Q', (double)'Y');
```

This is because the `x` and `Q` values are automatically promoted to ints (which are the same as longs for the sake of this example), and the integer values corresponding to `Z` and `Y` are cast to doubles.

To call `xyz` by using the `MODULEN` function, your attribute table must reflect the fact that you want to pass characters:

```
routine xyz minarg=2 maxarg=2 returns=long;
arg 1 input char byvalue format=$byval4.;
arg 2 input char byvalue format=$byval8.;
```

Note that it is important that the `BYVALUE` option appear in the `ARG` statement as well. Otherwise, `MODULEN` assumes that you want to pass a pointer to the routine, instead of a value.

Here is the `DATA` step that invokes `MODULEN` and passes it characters:

```
data _null_;
  x = moduln('xyz', 'X', 'Z');
  put x= ' (should be 1)';
  y = moduln('xyz', 'Q', 'Y');
  put y= ' (should be 2)';
run;
```

Understanding MODULE Log Messages

If you specify `i` in the control string parameter to `MODULE`, the SAS System prints several informational messages to the log. You can use these messages to determine whether you have passed incorrect arguments or coded the attribute table incorrectly.

Consider this example that uses `MODULEIN` from within the `IML` procedure. It uses the `MODULEIN` function to invoke the `changi` routine (stored in the theoretical `TRYMOD.DLL`). In the example, `MODULEIN` passes the constant 6 and the matrix `x2`,

which is a 4x5 matrix that is to be converted to an integer matrix. The attribute table for **changi** is as follows:

```
routine changi module=trymod returns=long;
arg 1 input num format=ib4. byvalue;
arg 2 update num format=ib4.;
```

The following IML step invokes MODULEIN:

```
proc iml;
  x1 = J(4,5,0);
  do i=1 to 4;
    do j=1 to 5;
      x1[i,j] = i*10+j+3;
    end;
  end;
  y1= x1;
  x2 = x1;
  y2 = y1;
  rc = modulein('*i','changi',6,x2);
```

The **'*i'** control string causes the lines shown in Output 10.1 on page 179 to be printed in the log.

Output 10.1 MODULEIN Output

```
---PARAM LIST FOR MODULEIN ROUTINE--- CHR PARM 1 885E0AA8 2A69 (*i)
CHR PARM 2 885E0AD0 6368616E6769 (changi)
NUM PARM 3 885E0AE0 0000000000001840
NUM PARM 4 885E07F0
0000000000002C400000000000002E40000000000003040000000000003140000000000003240
00000000000038400000000000003940000000000003A40000000000003B40000000000003C40
00000000000041400000000000080414000000000
---ROUTINE changi LOADED AT ADDRESS 886119B8 (PARMLIST AT 886033A0)--- PARM 1 06000000 <CALL-BY-VALUE>
PARM 2 88604720
0E0000000F00000010000000110000001200000018000000190000001A0000001B0000001C000000
22000000230000002400000025000000260000002C0000002D0000002E0000002F00000030000000
---VALUES UPON RETURN FROM changi ROUTINE--- PARM 1 06000000 <CALL-BY-VALUE>
PARM 2 88604720
140000001F0000002A0000003500000040000000820000008D000000980000000A3000000AE000000
F000000FB00000006010000110100001C0100005E01000069010000740100007F0100008A010000
---VALUES UPON RETURN FROM MODULEIN ROUTINE--- NUM PARM 3 885E0AE0 0000000000001840
NUM PARM 4 885E07F0
00000000000034400000000000003F40000000000004540000000000804A40000000000005040
0000000000406040000000000A0614000000000006340000000000606440000000000C06540
000000000006E40000000000606F4000000000
```

The output is divided into four sections.

①

The first section describes the arguments that are passed to MODULEIN.

The 'CHR PARM *n*' portion indicates that character parameter *n* was passed. In the example, 885E0AA8 is the actual address of the first character parameter that was passed to MODULEIN. The value at the address is hex 2A69, and the ASCII representation of that value ('*i') is in parentheses after the hex value. The second parameter is likewise printed. Only these first two arguments have their ASCII equivalents printed; this is because other arguments might contain unreadable binary data.

The remaining parameters appear with only hex representations of their values (NUM PARM 3 and NUM PARM 4 in the example).

The third parameter to MODULEIN is numeric, and it is at address 885E0AE0. The hex representation of the floating point number 6 is shown. The fourth parameter is at address 885E07F0, which points to an area that contains all the values for the 4x5 matrix. The *i option prints the entire argument; be careful if you use this option with large matrices because the log might become quite large.

②

The second section of the log lists the arguments to be passed to the requested routine and, in this case, changed. This section is important for determining if the arguments are being passed to the routine correctly. The first line of this section contains the name of the routine and its address in memory. It also contains the address of the location of the parameter block that MODULEIN created.

The log contains the status of each argument as it is passed. For example, the first parameter in the example is call-by-value (as indicated in the log). The second parameter is the address of the matrix. The log shows the address, along with the data to which it points.

Note that all the values in the first parameter and in the matrix are long integers because the attribute table states that the format is IB4.

③

In the third section, the log contains the argument values upon return from **changi**. The call-by-value argument is unchanged, but the other argument (the matrix) contains different values.

④

The last section of the log output contains the values of the arguments as they are returned to the MODULEIN calling routine.

Examples

Updating a Character String Argument

This example uses the Win32 routine GetTempPathA. This routine expects as an argument a pointer to a buffer, along with the length of the buffer. GetTempPathA fills the buffer with a null-terminated string representing the temporary path. Here is the C prototype for the GetTempPathA routine:

```
DWORD WINAPI GetTempPathA
    (DWORD nBufferLength, LPSTR lpBuffer);
```

Here is the attribute table:

```
routine GetTempPathA
    minarg=2
```



```

maxarg=2
stackpop=called
returns=long;
arg 1 input byvalue format=pib4.;
arg 2 update          format=$cstr200.;

```

Note that the STACKPOP=CALLED option is used; all Win32 service routines require this attribute. The first argument is passed by value because it is an input argument only. The second argument is an update argument because the contents of the buffer are to be updated. The \$CSTR200. format allows for a 200-byte character string that is null-terminated.

Here is the SAS code to invoke the function. In this example, the DLL name (KERNEL32) is explicitly given in the call (because the MODULE attribute was not used in the attribute file):

```

filename sascbtbl "sascbtbl.dat";
data _null_;
  length path $200;
  n = modulen( "*ie",
              "KERNEL32,GetTempPathA", 199, path );
  put n= path=;
run;

```

Note: KERNEL32.DLL is an internal DLL that is provided by OS/2. Its routines are described in the Microsoft Win32 SDK. Δ

The code produces these log messages:

```

NOTE: Variable PATH is uninitialized.
N=7 PATH=C:\TEMP

```

The example uses 199 as the buffer length because PATH can hold up to 200 characters with one character reserved for the null terminator. The \$CSTR200. informat ensures that the null-terminator and all subsequent characters are replaced by trailing blanks when control returns to the DATA step.

Passing Arguments by Value

This example calls the Beep routine, part of the Win32 API in the KERNEL32 DLL. Here is the C prototype for Beep:

```

BOOL Beep(DWORD dwFreq, DWORD dwDuration)

```

Here is the attribute table to use:

```

routine Beep
  minarg=2
  maxarg=2
  stackpop=called
  callseq=byvalue
  module=kernel32;
arg 1 num format=pib4.;
arg 2 num format=pib4.;

```

Because both arguments are passed by value, the example includes the CALLSEQ=BYVALUE attribute in the ROUTINE statement, so it is not necessary to specify the BYVALUE option in each ARG statement.

Here is the sample SAS code that used to call the Beep function:

```
filename sascbtbl 'sascbtbl.dat';
data _null_;
  rc = modulen("Beep",1380,1000);
run;
```

The computer speaker beeps.

Using PEEKC to Access a Returned Pointer

The following example uses the `lstrcat` routine, part of the Win32 API in `KERNEL32.DLL`. `lstrcat` accepts two strings as arguments, concatenates them, and returns a pointer to the concatenated string. The C prototype is

```
LPTSTR lstrcat (LPTSTR lpszString1,
               LPCTSTR lpszString2);
```

The following is the proper attribute table:

```
routine lstrcat
  minarg=2
  maxarg=2
  stackpop=called
  module=KERNEL32
  returns=ulong;
arg 1 char format=$cstr200.;
arg 2 char format=$cstr200.;
```

To use `lstrcat`, you need to use the SAS PEEKC function to access the data that are referenced by the returned pointer. Here is the sample SAS program that accesses `lstrcat`:

```
filename sascbtbl 'sascbtbl.dat';
data _null_;
  length string1 string2 conctstr $200;
  string1 = 'This is';
  string2 = ' a test!';
  rc = modulen('lstrcat',string1,string2);
  conctstr = peekc(rc,200);
  put conctstr=;
run;
```

The following output appears in the log:

```
conctstr=This is a test!
```

Upon return from `MODULEN`, the pointer value is stored in `RC`. The example uses the `PEEKC` function to return the 200 bytes at that location, and it uses the `$CSTR200.` format to produce a blank-padded string that replaces the null termination.

For more information about the PEEK functions, see “PEEK” on page 257.

Using Structures

“Grouping SAS Variables as Structure Arguments” on page 173 describes how to use the `FDSTART` attribute to pass several arguments as one structure argument to a DLL routine. See that section for an example of the `GetClientRect` attribute table and C language equivalent. This example shows how to invoke the `GetClientRect` function after you define the attribute table.

The most straightforward method is valid, but it generates a warning message about uninitialized variables:

```
filename sasbtbl 'sasbtbl.dat';
data _null_;
  hwnd=modulen('GetActiveWindow');
  call module('GetClientRect',hwnd,
    left,top,right,bottom);
  put _all_;
run;
```

To remove the warning, you can use the RETAIN statement to initialize the variables to 0. Also, you can use shorthand to specify the variable list in the MODULEN statement:

```
data _null_;
  retain left top right bottom 0;
  hwnd=modulen('GetActiveWindow');
  call module('GetClientRect',hwnd,
    of left--bottom);
  put _all_;
run;
```

Note that the OF keyword indicates that a list of variables follows, in this case delimited by the double-dash. The output in the log varies depending on the active window and looks something like the following:

```
HWND=3536768 LEFT=2 TOP=2 RIGHT=400
BOTTOM=587
```

Invoking a DLL Routine from PROC IML

This example shows how to pass a matrix as an argument within PROC IML. The example creates a 4x5 matrix. Each cell is set to $10x+y+3$, where x is the row number and y is the column number. For example, the cell at row 1 column 2 is set to $(10*1)+2+3$, or 15.

The example invokes several routines from the theoretical TRYMOD DLL. It uses the **changd** routine to add $100x+10y$ to each element, where x is the C row number (0 through 3) and y is the C column number (0 through 4). The first argument to **changd** indicates what extra amount to sum. The **changdx** routine executes the same way as **changd**, except that it expects a transposed matrix. The **changi** routine executes the same way as **changd** except that it expects a matrix of integers. The **changix** routine executes the same way as **changdx** except that integers are expected.

Note: A maximum of three arguments can be sent when you invoke a DLL routine from PROC IML. Δ

In this example, all four matrices $x1$, $x2$, $y1$, and $y2$ should become set to the same values after their respective MODULEIN calls. Here are the attribute table entries:

```
routine changd module=trymod returns=long;
arg 1 input num format=rb8. byvalue;
arg 2 update num format=rb8.;
routine changdx module=trymod returns=long
  transpose=yes;
arg 1 input num format=rb8. byvalue;
arg 2 update num format=rb8.;
routine changi module=trymod returns=long;
```

```

arg 1 input num format=ib4. byvalue;
arg 2 update num format=ib4.;
routine changix module=trymod returns=long
  transpose=yes;
arg 1 input num format=ib4. byvalue;
arg 2 update num format=ib4.;

```

Here is the PROC IML step:

```

proc iml;
  x1 = J(4,5,0);
  do i=1 to 4;
    do j=1 to 5;
      x1[i,j] = i*10+j+3;
    end;
  end;
  y1= x1; x2 = x1; y2 = y1;
  rc = modulein('changd',6,x1);
  rc = modulein('changdx',6,x2);
  rc = modulein('changi',6,y1);
  rc = modulein('changix',6,y2);
  print x1 x2 y1 y2;
run;

```

The following are the results of the PRINT statement:

X1				
20	31	42	53	64
130	141	152	163	174
240	251	262	273	284
350	361	372	383	394
X2				
20	31	42	53	64
130	141	152	163	174
240	251	262	273	284
350	361	372	383	394
Y1				
20	31	42	53	64
130	141	152	163	174
240	251	262	273	284
350	361	372	383	394
Y2				
20	31	42	53	64
130	141	152	163	174
240	251	262	273	284
350	361	372	383	394

The correct bibliographic citation for this manual is as follows: SAS Institute Inc., *SAS® Companion for the OS/2® Environment, Version 8*, Cary, NC: SAS Institute Inc., 1999. 448 pp.

SAS® Companion for the OS/2® Environment, Version 8

Copyright © 1999 by SAS Institute Inc., Cary, NC, USA.

ISBN 1-58025-521-3

All rights reserved. Produced in the United States of America. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, or otherwise, without the prior written permission of the publisher, SAS Institute Inc.

U.S. Government Restricted Rights Notice. Use, duplication, or disclosure of the software and related documentation by the U.S. government is subject to the Agreement with SAS Institute and the restrictions set forth in FAR 52.227-19 Commercial Computer Software-Restricted Rights (June 1987).

SAS Institute Inc., SAS Campus Drive, Cary, North Carolina 27513.

1st printing, October 1999

SAS® and all other SAS Institute Inc. product or service names are registered trademarks or trademarks of SAS Institute Inc. in the USA and other countries.® indicates USA registration.

IBM® and OS/2® are registered trademarks or trademarks of International Business Machines Corporation. ® indicates USA registration.

Other brand and product names are registered trademarks or trademarks of their respective companies.

The Institute is a private company devoted to the support and further development of its software and related services.