TECHNIQUES FOR INTERFACING SAS SOFTWARE WITH OTHER SOFTWARE

John Merline, Northwestern Mutual Life

ABSTRACT

This paper reflects my experiences in two areas:

1) Writing an interface to CINCOM SYSTEMS Data Base Management System, TOTAL.
2) Adding a Write To Operator (WTO) facility to the SAS Data step.

These techniques are applicable to interfacing SAS with any vendor's software. Topics covered include addressing SAS variables, using SAS Formats, reducing interface overhead and using SAS macros, along with general notes, references and a contact.

INTRODUCTION

I will assume that you have a working knowledge of the SAS base system and an elementary knowledge of assembler. In all of the assembler examples, the following equates will be used:

FR0 EQU 0 Floating point register 0
R2 EQU 2 General register 2
R3 EQU 3 General register 3
R4 EQU 4 General register 4
R7 EQU 7 General register 7
R8 EQU 8 General register 8
R9 EQU 9 General register 9
R10 EQU 10 General register 10
R12 EQU 12 General register 12

The SAS Institute supplies a well documented way to extend SAS by writing your own PROC. The language used in writing PROCs is PL/I, which is convenient considering that we are a PL/I shop. However, there are built in limitations to how PROC steps can be used. When interfacing to a DBMS in a PROC step, the most one could hope for is a generalized utility to extract information from the database and place it in a SAS dataset. While there are advantages (such as data independence) to this method over reading the database as a flat file in a DATA NULL step, it can hardly be called an interface.

The other way SAS supports extensions is through the CALL statement used in a DATA step. The CALL statement has the flexibility necessary to navigate the database based on values determined as the program runs, but unfortunately this requires an assembler interface to translate SAS internal data structures into a standard IBM calling sequence. This interface must also handle the mapping of the data returned into the appropriate SAS variables, converting all numeric data into the floating point form that SAS uses internally.

Such an interface can be used as a bridge to any product that supports the standard calling sequence. As each SAS Data Step is compiled and executed, CALL statements along with external FORMATS and INFORMATS (ones found on the format library) generate external references that cause them to be linked with the DATA step. When the DATA step runs, it will pass control to the interface when the CALL statement is executed. For conceptual overview of how and where an interface such as SAS/TOTAL fits into the SAS system is needed, see Figure 1.

ADDRESSING SAS VARIABLES

The first problem to dealt with is how to address the SAS variables passed as parameters on the CALL statement. On entry to a program called from a SAS data step, register twelve contains the address of the SAS communications area ($SASCOM), described by the SASCOM macro found in SAS.MACLIB. Register one contains the address of the parameter list (see Figure 4.). $SASCOM is used throughout the execution of a SAS job and is accessible to all tasks that run under the SAS supervisor.

There are 224 bytes of $SASCOM that are available for user programs. Figure 2 shows how the SAS/TOTAL uses this area ($USERCOM).

Among other things, $SASCOM contains the current values of all SAS system options and at least two important addresses. The first is the address of the program work area (@PROG) and is described by the SASPROG macro in SAS.MACLIB. The second is the address of the program data vector (PDV), where the SAS variables are found during execution of the DATA step.

Note that it is apparently a SAS coding convention that all labels that begin with '@' are found in the @PROG dsect and all labels beginning with a '$' belong to the $SASCOM dsect.

Like PL/I, SAS does not address
variables directly. For each variable in a SAS data step, there is a name descriptor. Each name descriptor is 32 bytes long and can be best described by moving it to the @NAME label in the @PROG dsect (see Figure 3).

One of the fields in the name descriptor is @OFF, which contains the offset from the beginning of the PDV to the SAS variable. Add this offset to the address of the PDV (the second address in the $CSADDR field) to address the SAS variable. The next step is locating the name descriptor for the SAS variable you want to access.

The parameter list that register one points to contains offsets into the name descriptor list (@NAMLIST). There is an offset for every variable that is specified on the SAS CALL statement. Since @NAMLIST begins with an '8', it must be found in the @PROG dsect. All that is needed to find the value of a SAS variable is to use register twelve to point to $SASCOM. Figure 4 summarizes the relationship of some of the control blocks used in addressing a SAS variable. Note that all the offsets are base ten.

Following is the sequence of code that illustrates addressing SAS variables. Comments (*) are included between the instructions for further explanation.

* The following USING is generated if the SASRENT macro is used.
USING $SASCOM,R12 addressability

* Use $CSADDR to point to $PROG (see Figure 2).
LM R9,R10,$CSADDR R9 --> $PROG, R10 --> PDV
USING @PROG,R9 adderssability
L R8,$NAMLIST R8 --> @NAMLIST

* Use R2 to point to the list of offsets (one for each variable) into the name descriptor list.
L R2,0(R7) offset into @NAMLIST

* Add that offset to @NAMLIST to find the name descriptor.
AR R2,R8 add address to offset

* Copy the name descriptor to the @PROG Dsect (see Figure 3).
MVC @NAME(@NAMEL),0(R2)

* Add the offset at @OFF to the address of the PDV ($CSADDR+4).

* Use the length of the SAS variable to build a parameter list.
XR R3,R3 zero R3
IC R3,$LNG length of SAS var
STH R3,$HWORD store in halfword

* Test whether the variable is character or numeric (remember that floating point is the only internal format for numeric variables in SAS).
TM @SW,$SWCHAR character var?
BZ CNSL070 NO.

* Convert a numeric variable to a more useful form such as binary.
LD FR0,0(R2) load numeric value
AW FR0,=XL8'4E00000010000000' store value

USING SAS FORMATS

The next thing an interface has to deal with is how to convert from the data formats SAS uses internally to external ones and back again. The most flexible way is to use the FORMAT-INFORMAT mechanism. This way, new data formats are supported automatically.

For the converting to work, an identical FORMAT and INFORMAT must be specified for the variable to be converted so that all the fields in @PROG are filled in by the SAS supervisor. As documented in the SAS Programmers' Guide, FORMATS and INFORMATs have calling conventions. Follow those conventions and you should be able to use any FORMAT-INFORMAT pair. This technique will not work if there is no corresponding FORMAT for a given INFORMAT and vice versa.

The following illustrates the conventions for numeric INFORMATs. Additional comments (*) are included between the assembler statements.

* Load R2 with the length of the SAS variable less one.
MVC @NAME(@NAMEL),0(R5)
IC R2,@FORL length of variable
CTR R2,0 length less one

* Load R4 with the number of decimal places.
SR R4,R4 zero R4
IC R4,$FORD decimal places

366
* Load R15 with the address of the
INFORMAT and branch to it.

L R15,@IFORA addr of informat
BALR R14,R15 branch to it

* The INFORMAT routine will branch back
to the address in R14 if successful,
or four plus the address if
unsuccessful (i.e. invalid packed
data, etc).

B TOTR540 informat successful
LD FR0,=X'8000000000000000'

When the subroutine returns control,
floating point register one will contain
the SAS internal value which you must
then place in the appropriate spot of
the PDV.

LH R6,60FF offset into PDV
AR R6,R10 add addr to offset
STD FR0,$DW put in double word
MVC 0(8,R6),$DW copy into PDV

The calling conventions for character
INFORMATs are the same except that R3
is set to the length of the variable and R4
is zeroed out to indicate no decimal
places. Since character variables can
have different lengths, use an execute
(EX) of a move character (MVC) to place
the value in the PDV.

The difference in calling FORMATs vs.
INFORMATs is that register fifteen is
loaded from @FORA instead of @IFORA.
Note that the output of a FORMAT is
always character. You must zero
register three prior to inserting the
width of the format each time you call
a numeric FORMAT because numeric FORMAT
routines alter the contents of register
three. FORMATs and INFORMATs are only
required to save registers five through
eleven.

REDUCING INTERFACE OVERHEAD

The first time the SAS/TOTAL interface
is called it obtains a storage area
(see Figure 5) used to hold an I/O
buffer from which database records are
read or written. This can be viewed as
an area in which the external forms of
SAS variables that might need
converting are kept. The parameter
list for each unique call to TOTAL and
name descriptors of the SAS variables
used in that call are also placed in
this storage area. The name
descriptors are used by SAS/TOTAL to
convert the fields in the I/O buffer
into the proper internal SAS format
using the SAS FORMAT-INFORMAT
mechanism.

The parmlist and name descriptors are
added to the working area starting from
the top down. The I/O buffer is placed
at the very bottom and working up from
that the information necessary to
process previous unique TOTAL calls are
stored. The information for the
current command is kept in $USERCOM.
When SAS/TOTAL is called, $USERCOM is
checked to see if the current command
is being repeated. If not, the working
storage area is checked to see if this
command has been done before.

If a match is found, the previously
constructed parmlist, SAS name
descriptors and I/O buffer is used.
Otherwise, the current command entry is
saved in working storage and a new
parmlist, list of name descriptors and
current command entry is built. When a
new entry needs to be built and there
is no more room, SAS/TOTAL abends.
Note that only command entries for
TOTAL commands that are repetitive by
nature (i.e. database reads or writes)
are saved.

USING SAS MACROS

Like SAS/TOTAL, the Write To Operator
interface uses SAS macros both to do
some validation of the input parameters
and to make the interface easier
to use. The Write To Operator macro
has a format of:

WRITOPER (MSG,TEST);

where MSG is either the name of the SAS
variable (up to 80 characters) or a
literal enclosed in single quotes. If
MSG is missing, it defaults to the SAS
variable _MSG_. If a literal is used,
the value of the literal is assigned to
the SAS variable _MSG_ , where $ is a
unique number assigned at each
invocation of the macro.

TEST is an optional parameter that
changes the Write To Operator to a
Write To Programmer. In foreground,
this option will route the message to
your terminal instead of the operator
if your TSO profile is set properly.

To enable this function, use the
PROFILE TSO command to set WTPMSG.
This has to be done once. If you are
running in background, the message will
appear in the JES messages of the JOB
log and will not be sent to the
operator.

The Write To Operator macro first
determines if a literal was specified. If
so, the literal is assigned to a
temporary variable and the variable
passed in the CALL statement. The CALL statement does not support literal parameters. The macro variables all begin with two underscores to lessen the chance that someone might use that variable name on a %LET statement.

If the first parameter on the macro is missing the SAS variable _MSG it is assumed to contain the message and is passed in the CALL statement. The DO and END statements allows the macro to be invoked from the THEN clause of an IF.

The following is a listing of the Write To Operator macro with a few examples of how to use it.

```sas
%MACRO WRITOPER(__MSG, __ROUTCD);
DO;
  %IF __MSG EQ %THEN %LET __MSG = __MSG ;
  %IF %INDEX(& __MSG,%QUOTE(%STR(%'») GT 0 %THEN %DO;
    %UNQUOTE(MSG&SYSINDEXSTR() = & __MSG%STR( ; ) ) ;
  %LET __MSG = __MSG&SYSINDEXSTR(_);%END;
  %IF & ROUTCD EQ TEST %THEN %DO;
    %UNQUOTE(CALL SASCNSL (& __MSG,_ROUTCD%STR( ; ) );
  %ELSE %UNQUOTE(CALL SASCNSL (& __MSG)%STR( ; ) );
%END;
%MEND WRITOPER;

DATA _NULL_;
%WRITOPER('TESTING NEW FUNCTION',TEST);

DATA _NULL_;
/* debugging a do loop in foreground */
DO I=1 TO 10;
  IF MOD(I,2) = 0 THEN DO;
    MSG = 'THE VALUE OF I IS ' || I;
    %WRITOPER(MSG,TEST);
  END;
END;

DATA BIGJOB;
  MSG = 'SOMETHING TERRIBLE HAPPENED!';
  _ERROR = 1;
  IF _ERROR THEN %WRITOPER;

GENERAL NOTES
The following notes are pieces of information to keep in mind:

1) There is currently no way to make sure that the temporary variables created by SAS macros are not output to the SAS dataset. I am afraid of creating a DROP statement because of possible conflicts with user KEEP statements. SAS has a way of declaring temporary variables such as _N as pseudo variables so that they do not get output, but this is only available at DATA step compile time.

One possible solution is to follow the SAS convention for temporary variables of starting and ending the variable name with an underscore. I have coded a SAS macro that using PROC TRANSPOSE, will remove these temporary variables after the SAS dataset is created, producing a new, clean dataset.

2) If you use the SASRENT macro to establish addressability and allocate a DSA (dynamic save area, aḷa PL/1) as I do, you have to restore register one from your caller’s save area. $SASRENT does not preserve the contents of register one.

3) The SAS user exit facilities can be used to clean up the working storage area after each data step if problems develop due to saving a large number of unique calls as a means of reducing the overhead associated with calling the interface.

REFERENCES
Technical Report P-127: Enhancements to the SAS Macro Facility
Assembler H Version 2 Application Programming: Language Reference, IBM Publication number GC26-4037-0
IBM System/370 Extended Architecture Principles of Operation, IBM Publication number SA22-7085-0

CONTACT
Northwestern Mutual Life
720 East Wisconsin Avenue
Milwaukee, Wisconsin 53202
ATTN: John Merline

(414) 271-1444 Ext. 4845
Figure 1

CINCOM TOTAL INTERFACE (DATBASE) → LOADS → SASTOT (LINKED WITH THE DATA STEP AND RESIDENT DURING THE DATA STEP) → WORKING STORAGE (GETAINED BY SASTOT AT SIMON) → PROGRAM DATA VECTOR

Figure 2

$SASCOM (pointed to by R12 on entry)

+41C $CSADDR
(the first two addresses area for the $PROG and PDV respectively.)

$USERCOM

+900
+0 $MODULE
+8 $TOTALINF
+C $TSTATUS
+10 $NXTWORD
+18 $STOP
+1C $STOP
+28 $START
+2C $BUFFADDR
+28 $CURRCMD EQU *
+28 $CMD_ID
+2B $CMDTYPE
+2C $PARHADR
+30 $VARADR
+34 $VSTATE
+38 $VLENGTH

Figure 3

$PROG (pointed to by $CSADDR)

+148 $CHNLIST addr of name descriptors

A mapping of one descriptor

+1B0
+0 $NAME
+6 $RECCODE
+8 $LSL
+9 $LSNG
+A $SM
+B $SMZ
+C $LSNGSPC
+D $FORL
+8 $FORF
+F $FORJ
+10 $LBLEN
+11 $FOREND
+12 $FOREND
+13 slack byte
+14 $OFF
+16 $LBLOFF
+18 $FORA
+1C $FORA
$NAME EQU $NAME

NOTE: All offsets are given in hexadecimal.
Figure 4

Figure 5

370