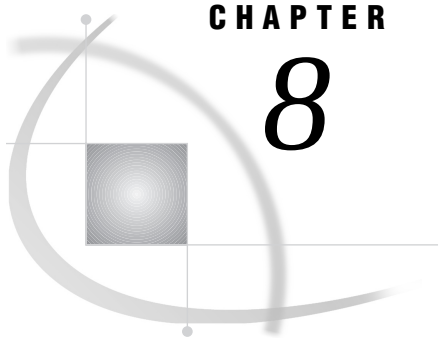

CHAPTER

8

Interfaces with the Macro Facility

Introduction **91**
DATA Step Interfaces **92**
CALL EXECUTE Routine Timing Details **92**
A Simple Example of CALL EXECUTE Timing **92**
Another Example Using CALL EXECUTE **93**
Using SAS Language Functions in the DATA Step and Macro Facility **95**
Interfaces with the SQL Procedure **96**
INTO Clause **96**
Controlling Job Execution **96**
Interfaces with Screen Control Language **97**
Understanding How Macro References Are Resolved by SCL **97**
Referencing Macro Variables in Submit Blocks **98**
Considerations for Sharing Macros between SCL Programs **98**
Example Using Macros in an SCL Program **98**
SAS/CONNECT Interfaces **99**
Example Using %SYSRPUT to Check the Value of a Return Code on a Remote Host **100**

Introduction

An *interface* with the macro facility is not part of the macro processor but rather a SAS software feature that enables another portion of the SAS language to interact with the macro facility during execution. For example, a DATA step interface enables you to access macro variables from the DATA step. Macro facility interfaces are useful because, in general, macro processing happens before DATA step, SQL, SCL, or SAS/CONNECT execution, so the connection between the macro facility and the rest of the SAS System is not usually dynamic. But by using an interface to the macro facility, you can dynamically connect the macro facility to the rest of the SAS System.

Note: The %SYSFUNC and %QSYSFUNC macro functions enable you to use SAS language functions with the macro processor. The %SYSCALL macro statement enables you to use SAS language CALL routines with the macro processor. While these elements of the macro language are not considered true macro facility interfaces, they are discussed in this chapter. See Chapter 13, “Macro Language Dictionary,” for more information on these macro language elements. △

While this chapter includes some examples, you can find additional examples for each item in Chapter 13.

DATA Step Interfaces

DATA step interfaces consist of four tools that enable a program to interact with the macro facility *during* DATA step execution. Because the work of the macro facility takes place before DATA step execution begins, information provided by macro statements has already been processed during DATA step execution. Use one of the DATA step interfaces to interact with the macro facility during DATA step execution. Use DATA step interfaces to

- pass information from a DATA step to a subsequent step in a SAS program
- invoke a macro based on information available only when the DATA step executes
- resolve a macro variable while a DATA step executes.

Table 8.1 on page 92 lists the DATA step interfaces by category and their uses.

Table 8.1 DATA Step Interfaces

Category	Tool	Description
Execution	CALL EXECUTE routine	resolves its argument and executes the resolved value at the next step boundary (if the value is a SAS statement) or immediately (if the value is a macro language element)
Resolution	RESOLVE function	resolves the value of a text expression during DATA step execution
Read or Write	SYMGET function	returns the value of a macro variable during DATA step execution
	CALL SYMPUT routine	assigns a value produced in a DATA step to a macro variable

CALL EXECUTE Routine Timing Details

CALL EXECUTE is useful when you want to execute a macro conditionally. But you must remember that if CALL EXECUTE produces macro language elements, those elements execute immediately; if CALL EXECUTE produces SAS language statements, or if the macro language elements generate SAS language statements, those statements execute after the end of the DATA step's execution.

Note: Because macro references execute immediately and SAS statements do not execute until after a step boundary, you cannot use CALL EXECUTE to invoke a macro that contains references for macro variables that are created by CALL SYMPUT in that macro. △

Here are two examples that illustrate the timing problems that users frequently have with CALL EXECUTE.

A Simple Example of CALL EXECUTE Timing

In this example, the CALL EXECUTE routine is used incorrectly:

```
data prices;  /* ID for price category and actual price */
   input code amount;
```

```
   cards;
56 300
99 10000
24 225
;

%macro items;
   %global special;
   %let special=football;
%mend items;

data sales;    /* incorrect usage */
   set prices;
   length saleitem $ 20;
   call execute('%items');
   saleitem="&special";
run;
```

In the DATA SALES step, the assignment statement for SALEITEM requires the value of the macro variable SPECIAL at DATA step compilation. CALL EXECUTE does not produce the value until DATA step execution. Thus, you receive a message about an unresolved macro variable, and the value assigned to SALEITEM is **&special**.

In this example, it would be better to eliminate the macro definition (the %LET macro statement is valid in open code) or move the DATA SALES step into the macro ITEMS. In either case, CALL EXECUTE is not necessary or useful. Here is one version of this program that works:

```
data prices;   /* ID for price category and actual price */
   input code amount;
   cards;
56 300
99 10000
24 225
;

%let special=football;  /* correct usage */

data sales;
   set prices;
   length saleitem $ 20;
   saleitem="&special";
run;
```

The %GLOBAL statement isn't necessary in this version. Because the %LET statement is executed in open code, it automatically creates a global macro variable. (See Chapter 5, "Scope of Macro Variables," for more information about macro variable scope.)

Another Example Using CALL EXECUTE

This example shows a common pattern that causes an error.

```
/* This version of the example shows the problem. */

data prices;        /* ID for price category and actual price */
   input code amount;
   cards;
```

```
56 300
99 10000
24 225
;
data names;     /* name of sales department and item sold */
   input dept $ item $;
   cards;
BB  Boat
SK  Skates
;

%macro items(codevar=);  /* create macro variable if needed */
   %global special;
   data _null_;
      set names;
      if &codevar=99 and dept='BB' then call symput('special', item);
   run;
%mend items;

data sales;  /* attempt to reference macro variable fails */
   set prices;
   length saleitem $ 20;
   if amount > 500 then
      call execute('%items(codevar=' || code || ')' );
   saleitem="&special";
run;
```

In this example, the DATA SALES step still requires the value of SPECIAL during compilation. The CALL EXECUTE routine is useful in this example because of the conditional IF statement. But as in the first example, CALL EXECUTE still invokes the macro ITEMS during DATA step execution–not compilation. The macro ITEMS generates a DATA _NULL_ step that executes after the DATA SALES step has ceased execution. The DATA _NULL_ step creates SPECIAL, and the value of SPECIAL is available after the _NULL_ step ceases execution–much later than when the value was needed.

This version of the example corrects the problem:

```
/* This version solves the problem. */

data prices;     /* ID for price category and actual price */
   input code amount;
   cards;
56 300
99 10000
24 225
;

data names;     /* name of sales department and item sold */
   input dept $ item $;
   cards;
BB  Boat
SK  Ski
;
%macro items(codevar=);   /* create macro variable if needed */
   %global special;
```

```
   data _null_;
      set names;
      if &codevar=99 and dept='BB' then
         call symput('special', item);
   run;
%mend items;

data _null_;   /* call the macro in this step */
   set prices;
   if amount > 500 then
      call execute('%items(codevar=' || code || ')' );
run;

data sales;    /* use the value created by the macro in this step */
   set prices;
   length saleitem $ 20;
   saleitem="&special";
run;
```

This version uses one DATA _NULL_ step to call the macro ITEMS. After that step ceases execution, the DATA _NULL_ step generated by ITEMS executes and creates the macro variable SPECIAL. Then the DATA SALES step references the value of SPECIAL as usual.

Using SAS Language Functions in the DATA Step and Macro Facility

The macro functions %SYSFUNC and %QSYSFUNC can call SAS language functions and functions written with SAS/TOOLKIT software to generate text in the macro facility. %SYSFUNC and %QSYSFUNC have one difference: the %QSYSFUNC masks special characters and mnemonics and %SYSFUNC does not. For more information on these functions, see the %QSYSFUNC and %SYSFUNC topics in Chapter 13.

%SYSFUNC arguments are a single SAS language function and an optional format, as shown in the following examples:

```
%sysfunc(date(),worddate.)
%sysfunc(attrn(&dsid,NOBS))
```

You cannot nest SAS language functions within %SYSFUNC. However, you can nest %SYSFUNC functions that call SAS language functions, as in the following statement:

```
%sysfunc(compress(%sysfunc(getoption(sasautos)),%str(%)%('')))
```

This example returns the value of the SASAUTOS= system option, using the COMPRESS function to eliminate opening parentheses, closing parentheses, and single quotation marks from the result. Note the use of the %STR function and the unmatched parentheses and quotation marks that are marked with a percent sign (%).

All arguments in SAS language functions within %SYSFUNC must be separated by commas. You cannot use argument lists preceded by the word OF.

Because %SYSFUNC is a macro function, you do not need to enclose character values in quotation marks as you do in SAS language functions. For example, the arguments to the OPEN function are enclosed in quotation marks when the function is used alone but do not require quotation marks when used within %SYSFUNC.

Here are some examples of the contrast between using a function alone and within %SYSFUNC:

□ **dsid = open("sasuser.houses","i");**

- `dsid = open("&mydata","&mode");`
- `%let dsid = %sysfunc(open(sasuser.houses,i));`
- `%let dsid = %sysfunc(open(&mydata,&mode));`

You can use %SYSFUNC and %QSYSFUNC to call all of the DATA step SAS functions except DIF, DIM, HBOUND, INPUT, LAG, LBOUND, PUT, RESOLVE, and SYMGET. In the macro facility, SAS language functions called by %SYSFUNC can return values with a length up to 32K. However, within the DATA step, return values are limited to the length of a data set character variable.

The %SYSCALL macro statement enables you to use SAS language CALL routines with the macro processor, and it is described in Chapter 13.

Interfaces with the SQL Procedure

Structured Query Language (SQL) is a standardized, widely used language for retrieving and updating data in databases and relational tables. The SAS System's SQL processor enables you to

- create tables and views
- retrieve data stored in tables
- retrieve data stored in SQL and SAS/ACCESS views
- add or modify values in tables
- add or modify values in SQL and SAS/ACCESS views.

INTO Clause

SQL provides the INTO clause in the SELECT statement for creating SAS macro variables. You can create multiple macro variables with a single INTO clause. The INTO clause follows the same scoping rules as the %LET statement. See Figure 5.3 in Chapter 5 for a summary of how macro variables are created. For further details and examples relating to the INTO clause, see Chapter 13.

Controlling Job Execution

PROC SQL also provides macro tools to

- stop execution of a job if an error occurs
- execute programs conditionally based on data values.

Table 8.2 on page 97 provides information about macro variables created by SQL that affect job execution.

Table 8.2 Macro Variables that Affect Job Execution

Macro Variable	Description
SQLOBS	contains the number of rows or observations produced by a SELECT statement.
SQLRC	contains the return code from an SQL statement. For return codes, see SAS SQL documentation.
SQLOOPS	contains the number of iterations that the inner loop of PROC SQL processes.

Interfaces with Screen Control Language

You can use the SAS macro facility to define macros and macro variables for your SCL program. Then, you can pass parameters between macros and the rest of your program. Also, through the use of the autocall and compiled stored macro facilities, macros can be used by more than one SCL program.

Note: Macro modules can be more complicated to maintain than a program segment because of the symbols and macro quoting that may be required. Also, implementing modules as macros does not reduce the size of the compiled SCL code. Program statements generated by a macro are added to the compiled code as if those lines existed at that location in the program. △

Table 8.3 on page 97 lists the SCL macro facility interfaces.

Table 8.3 SCL Interfaces to the Macro Facility

Category	Tool	Description
Read or Write	SYMGET	returns the value of a global macro variable during SCL execution
	SYMGETN	returns the value of a global macro variable as a numeric value
	CALL SYMPUT	assigns a value produced in SCL to a global macro vairable
	CALL SYMPUTN	assigns a numeric value to a global macro variable

Note: It is inefficient to use SYMGETN to retrieve values that are not assigned with SYMPUTN. It is also inefficient to use **&** to reference a macro variable that was created with CALL SYMPUTN. Instead, use SYMGETN. In addition, it is inefficient to use SYMGETN and CALL SYMPUTN with values that are not numeric. △

For details on these elements, see Chapter 13.

Understanding How Macro References Are Resolved by SCL

An important point to remember when using the macro facility with SCL is that macros and macro variable references in SCL programs are resolved when the SCL

program compiles, not when you execute the application. To further control the assignment and resolution of macros and macro variables, use the following techniques:

- If you want macro variables to be assigned and retrieved when the SCL program executes, use CALL SYMPUT and CALL SYMPUTN in the SCL program.
- If you want a macro call or macro variable reference to resolve when an SCL program executes, use SYMGET and SYMGETN in the SCL program.

Referencing Macro Variables in Submit Blocks

In SCL, macro variable references are resolved at compile time unless they are in a Submit block. When SCL encounters a name prefixed with an ampersand (&) in a Submit block, it checks whether the name following the ampersand is the name of an SCL variable. If so, SCL substitutes the value of the corresponding variable for the variable reference in the submit block. If the name following the ampersand does not match any SCL variable, the name passes intact (including the ampersand) with the submitted statements. When the SAS System processes the statements, it attempts to resolve the name as a macro variable reference

To guarantee that a name is passed as a macro variable reference in submitted statements, precede the name with two ampersands (for example, &&DSNAME). If you have both a macro variable and an SCL variable with the same name, a reference with a single ampersand substitutes the SCL variable. To force the macro variable to be substituted, reference it with two ampersands (&&).

Considerations for Sharing Macros between SCL Programs

Sharing macros between SCL programs can be useful, but it can also raise some configuration management problems. If a macro is used by more than one program, you must keep track of all the programs that use it so you can recompile all of them each time the macro is updated. Because SCL is compiled, each SCL program that calls a macro must be recompiled whenever that macro is updated to update the program with the new macro code.

CAUTION:

Recompile the SCL program. If you fail to recompile the SCL program when you update the macro, you run the risk of the compiled SCL being out of sync with the source. △

Example Using Macros in an SCL Program

This SCL program is for an example application with the fields BORROWED, INTEREST, and PAYMENT. The program uses the macros CKAMOUNT and CKRATE to validate values entered into fields by users. The program calculates the payment, using values entered for the interest rate (INTEREST) and the sum of money (BORROWED).

```
/* Display an error message if AMOUNT */
   /* is less than zero or larger than 1000. */
%macro ckamount(amount);
   if (&amount < 0) or (&amount > 1000) then
      do;
         erroron borrowed;
         _msg_='Amount must be between $0 and $1,000.';
         stop;
      end;
```

```
   else erroroff borrowed;
%mend ckamount;

   /* Display an error message if RATE */
   /* is less than 0 or greater than 1.5 */
%macro ckrate(rate);
   if (&rate < 0) or (&rate > 1) then
      do;
         erroron interest;
         _msg_='Rate must be between 0 and 1.5';
         stop;
      end;
   else erroroff interest;
%mend ckrate;

   /*  Open the window with BORROWED at 0 and INTEREST at .5.  */
INIT:
   control error;
   borrowed=0;
   interest=.5;
return;

MAIN:
      /*  Run the macro CKAMOUNT to validate  */
      /*  the value of BORROWED.              */
   %ckamount(borrowed);
      /*  Run the macro CKRATE to validate  */
      /* the value of INTEREST.             */
   %ckrate(interest)
      /*  Calculate payment.  */
   payment=borrowed*interest;
return;

TERM:
return;
```

SAS/CONNECT Interfaces

The %SYSRPUT macro statement is submitted with SAS/CONNECT to a remote host to retrieve the value of a macro variable stored on the remote host. %SYSRPUT assigns that value to a macro variable on the local host. %SYSRPUT is similar to the %LET macro statement because it assigns a value to a macro variable. However, %SYSRPUT assigns a value to a variable on the local host, not on the remote host where the statement is processed. The %SYSRPUT statement places the macro variable in the current scope of the local host.

Note: The names of the macro variables on the remote and local hosts must not contain a leading ampersand. △

The %SYSRPUT statement is useful for capturing the value of the automatic macro variable SYSINFO and passing that value to the local host. SYSINFO contains return-code information provided by some SAS procedures. Both the UPLOAD and the DOWNLOAD procedures of SAS/CONNECT can update the macro variable SYSINFO and set it to a nonzero value when the procedure terminates due to errors. You can use

%SYSRPUT on the remote host to send the value of the SYSINFO macro variable back to the local SAS session. Thus, you can submit a job to the remote host and test whether a PROC UPLOAD or DOWNLOAD step has successfully completed before beginning another step on either the remote host or the local host.

To use %SYSRPUT, you must have invoked a remote SAS display manager session by submitting the DMR option with the SAS command. For details about using %SYSRPUT, see the SAS/CONNECT documentation.

To create a new macro variable or to modify the value of an existing macro variable on a remote host or a server, use the %SYSLPUT macro statement.

Example Using %SYSRPUT to Check the Value of a Return Code on a Remote Host

This example illustrates how to download a file and return information about the success of the step. When remote processing is completed, the job checks the value of the return code stored in RETCODE. Processing continues on the local host if the remote processing is successful. In this example, the %SYSRPUT statement follows a PROC DOWNLOAD step, so the value returned by SYSINFO indicates the success of the PROC DOWNLOAD step:

```
/* This code executes on the remote host. */
rsubmit;
   proc download data=remote.mydata out=local.mydata;
   run;
         /* RETCODE is on the local host. */
         /* SYSINFO is on the remote host. */
   %sysrput retcode=&sysinfo;
endrsubmit;

   /* This code executes on the local host. */
%macro checkit;
   %if &retcode = 0 %then
      %do;
         further processing on local host
      %end;
%mend checkit;

%checkit
```

To determine the success or failure of a step executed on a remote host, use the %SYSRPUT macro statement to check the value of the automatic macro variable SYSERR.

For more details and syntax of the %SYSRPUT statement, refer to Chapter 13.

The correct bibliographic citation for this manual is as follows: SAS Institute Inc., *SAS Macro Language: Reference, Version 8*, Cary, NC: SAS Institute Inc., 1999. 310 pages.

SAS Macro Language: Reference, Version 8

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

1-58025-522-1

All rights reserved. Printed in the United States of America. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, by 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 by the government is subject to restrictions as 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.

OS/2® is a registered trademark or trademark of International Business Machines Corporation.

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.