SAS® Macro Language: Reference

2020.1 - 2024.10*

* This document might apply to additional versions of the software. Open this document in SAS Help Center and click on the version in the banner to see all available versions.

SAS® Documentation
October 1, 2024

The correct bibliographic citation for this manual is as follows: SAS Institute Inc. 2020. *SAS® Macro Language: Reference*. Cary, NC: SAS Institute Inc.

SAS® Macro Language: Reference

SAS Institute Inc., SAS Campus Drive, Cary, NC 27513-2414

October 2024

v_001-P1:mcrolref

Contents

Syntax Conventions for the SAS Language

Overview of Syntax Conventions for the SAS Language

SAS uses standard conventions in the documentation of syntax for SAS language elements. These conventions enable you to easily identify the components of SAS syntax. The conventions can be divided into these parts:

- syntax components
- style conventions
- special characters
- references to SAS libraries and external files

Syntax Components

The components of the syntax for most language elements include a keyword and arguments. For some language elements, only a keyword is necessary. For other language elements, the keyword is followed by an equal sign (=). The syntax for arguments has multiple forms in order to demonstrate the syntax of multiple arguments, with and without punctuation.

keyword

specifies the name of the SAS language element that you use when you write your program. Keyword is a literal that is usually the first word in the syntax. In a CALL routine, the first two words are keywords.

In these examples of SAS syntax, the keywords are bold:

CHAR (*string, position*)

CALL RANBIN (*seed, n, p, x*);

ALTER (*alter-password*)

BEST *w*.

REMOVE <*data-set-name*>

In this example, the first two words of the CALL routine are the keywords:

CALL RANBIN(*seed*, *n*, *p*, *x*)

The syntax of some SAS statements consists of a single keyword without arguments:

DO;
... *SAS code* ...
END;

Some system options require that one of two keyword values be specified:

DUPLEX | **NODUPLEX**

Some procedure statements have multiple keywords throughout the statement syntax:

CREATE <UNIQUE> **INDEX** *index-name* **ON** *table-name* (*column-1* <, *column-2*, ...>)

argument
specifies a numeric or character constant, variable, or expression. Arguments follow the keyword or an equal sign after the keyword. The arguments are used by SAS to process the language element. Arguments can be required or optional. In the syntax, optional arguments are enclosed in angle brackets (< >).

In this example, *string* and *position* follow the keyword CHAR. These arguments are required arguments for the CHAR function:

CHAR (*string, position*)

Each argument has a value. In this example of SAS code, the argument *string* has a value of 'summer', and the argument *position* has a value of 4:

```
x=char('summer', 4);
```

In this example, *string* and *substring* are required arguments, whereas *modifiers* and *startpos* are optional.

FIND(*string, substring* <, *modifiers*> <, *startpos*>

argument(s)
specifies that one argument is required and that multiple arguments are allowed. Separate arguments with a space. Punctuation, such as a comma (,) is not required between arguments.

The MISSING statement is an example of this form of multiple arguments:

MISSING *character(s)*;

<LITERAL_ARGUMENT> *argument-1* <<LITERAL_ARGUMENT> *argument-2* ... >
specifies that one argument is required and that a literal argument can be associated with the argument. You can specify multiple literals and argument

pairs. No punctuation is required between the literal and argument pairs. The ellipsis (...) indicates that additional literals and arguments are allowed.

The BY statement is an example of this argument:

BY <DESCENDING> *variable-1* <<DESCENDING> *variable-2* ...>;

argument-1 <*options*> <*argument-2* <*options*> ...>
specifies that one argument is required and that one or more options can be associated with the argument. You can specify multiple arguments and associated options. No punctuation is required between the argument and the option. The ellipsis (...) indicates that additional arguments with an associated option are allowed.

The FORMAT procedure PICTURE statement is an example of this form of multiple arguments:

PICTURE name <(*format-options*)>
<*value-range-set-1* <(*picture-1-options*)>
<*value-range-set-2* <(*picture-2-options*)> ...>>;

argument-1=value-1 <*argument-2=value-2* ...>
specifies that the argument must be assigned a value and that you can specify multiple arguments. The ellipsis (...) indicates that additional arguments are allowed. No punctuation is required between arguments.

The LABEL statement is an example of this form of multiple arguments:

LABEL *variable-1=label-1* <*variable-2=label-2* ...>;

argument-1 <, *argument-2*, ...>
specifies that one argument is required and that you can specify multiple arguments that are separated by a comma or other punctuation. The ellipsis (...) indicates a continuation of the arguments, separated by a comma. Both forms are used in the SAS documentation.

Here are examples of this form of multiple arguments:

AUTHPROVIDERDOMAIN (*provider-1:domain-1* <, *provider-2:domain-2*, ...>

INTO :*macro-variable-specification-1* <, :*macro-variable-specification-2*, ...>

Note: In most cases, example code in SAS documentation is written in lowercase with a monospace font. You can use uppercase, lowercase, or mixed case in the code that you write.

Style Conventions

The style conventions that are used in documenting SAS syntax include uppercase bold, uppercase, and italic:

UPPERCASE BOLD
identifies SAS keywords such as the names of functions or statements. In this example, the keyword ERROR is written in uppercase bold:

ERROR <*message*>;

UPPERCASE
identifies arguments that are literals.

In this example of the CMPMODEL= system option, the literals include BOTH, CATALOG, and XML:

CMPMODEL=BOTH | CATALOG | XML |

italic
identifies arguments or values that you supply. Items in italic represent user-supplied values that are either one of the following:

- nonliteral arguments. In this example of the LINK statement, the argument *label* is a user-supplied value and therefore appears in italic:

  LINK *label*;

- nonliteral values that are assigned to an argument.

  In this example of the FORMAT statement, the argument DEFAULT is assigned the variable *default-format*:

  FORMAT *variable(s)* <*format* > <DEFAULT = *default-format*>;

Special Characters

The syntax of SAS language elements can contain the following special characters:

=
an equal sign identifies a value for a literal in some language elements such as system options.

In this example of the MAPS system option, the equal sign sets the value of MAPS:

MAPS=*location-of-maps*

< >
angle brackets identify optional arguments. A required argument is not enclosed in angle brackets.

In this example of the CAT function, at least one item is required:

CAT (*item-1* <, *item-2*, ...>)

|
a vertical bar indicates that you can choose one value from a group of values. Values that are separated by the vertical bar are mutually exclusive.

In this example of the CMPMODEL= system option, you can choose only one of the arguments:

CMPMODEL=BOTH | CATALOG | XML

...

an ellipsis indicates that the argument can be repeated. If an argument and the ellipsis are enclosed in angle brackets, then the argument is optional. The repeated argument must contain punctuation if it appears before or after the argument.

In this example of the CAT function, multiple *item* arguments are allowed, and they must be separated by a comma:

CAT (*item-1* <, *item-2*, ...>)

'*value*' or "*value*"

indicates that an argument that is enclosed in single or double quotation marks must have a value that is also enclosed in single or double quotation marks.

In this example of the FOOTNOTE statement, the argument *text* is enclosed in quotation marks:

FOOTNOTE <*n*> <*ods-format-options* '*text*' | "*text*">;

;

a semicolon indicates the end of a statement or CALL routine.

In this example, each statement ends with a semicolon:

```
data namegame;
   length color name $8;
   color = 'black';
   name = 'jack';
   game = trim(color) || name;
run;
```

References to SAS Libraries and External Files

Many SAS statements and other language elements refer to SAS libraries and external files. You can choose whether to make the reference through a logical name (a libref or fileref) or use the physical filename enclosed in quotation marks.

If you use a logical name, you typically have a choice of using a SAS statement (LIBNAME or FILENAME) or the operating environment's control language to make the reference. Several methods of referring to SAS libraries and external files are available, and some of these methods depend on your operating environment.

In the examples that use external files, SAS documentation uses the italicized phrase *file-specification*. In the examples that use SAS libraries, SAS documentation uses the italicized phrase *SAS-library* enclosed in quotation marks:

```
infile file-specification obs = 100;
libname libref 'SAS-library';
```

PART 1

Understanding and Using the Macro Facility

1

Introduction to the Macro Facility

Getting Started with the Macro Facility

This document is the macro facility language reference for SAS. It is a reference for the SAS macro language processor and defines the SAS macro language elements. This section introduces the SAS macro facility using simple examples and explanation.

The *macro facility* is a tool for extending and customizing SAS and for reducing the amount of text that you must enter to do common tasks. The macro facility enables you to assign a name to character strings or groups of SAS programming statements. You can work with the names that you created rather than with the text itself.

The SAS macro language is a string-based language. It does not support the use of hexadecimal character constants.

Note: The SAS macro language does not support using hexadecimal values to specify non-printable characters.

When you use a macro facility name in a SAS program or from a command prompt, the macro facility generates SAS statements and commands as needed. The rest of SAS receives those statements and uses them in the same way it uses the ones that you enter in the standard manner.

The macro facility has two components:

macro processor
: is the portion of SAS that does the work

macro language
: is the syntax that you use to communicate with the macro processor

When SAS compiles program text, two delimiters trigger macro processor activity:

&*name*
: refers to a macro variable. "Replacing Text Strings Using Macro Variables" on page 8 explains how to create a macro variable. The form &*name* is called a macro variable reference.

%*name*
: refers to a macro. "Generating SAS Code Using Macros" on page 9 explains how to create a macro. The form %*name* is called a macro call.

The text substitution produced by the macro processor is completed before the program text is compiled and executed. The macro facility uses statements and functions that resemble the statements and functions that you use in the DATA step. An important difference, however, is that macro language elements can enable only text substitution and are not present during program or command execution.

Note: Three SAS statements begin with a % that are not part of the macro facility. These elements are the %INCLUDE, %LIST, and %RUN statements in *SAS DATA Step Statements: Reference*.

Using the Macro Facility on the SAS Viya Platform

The macro facility does not run within Cloud Analytic Services (CAS). The macro facility runs in the Compute Server. You can use the macro facility to generate code for procedures, DATA steps, and global statements. Code generated by the macro facility can use LIBNAME engines. CAS uses caslibs and data connectors instead of LIBNAME statements.

All macros are created in and executed from the Compute Server. In this example the macro myMac1 contains a DATA step to add the X variable to the Work.Class data set. The DATA step created by myMac1 macro reads the data set Class from the SASHELP library and outputs a data set in the Work directory.

For the myMac2 macro, a CAS LIBNAME statement is entered, which assigns a caslib named Mycas. The macro myMac2 runs in the Compute Server to add the Y variable and outputs to the Mycas.Class table.

```
/* Macro 1 */
%macro myMac1;
    data class;
       set sashelp.class;
       x=1;
       put "**** ";
       put " This macro is running in the Compute Server and";
       put " creating a SAS data set in the Work library.";
       put "**** ";
       put _hostname_ 'thread #' _threadid_;
    run;
%mend myMac1;
%myMac1;

/* Specify the CAS host and port, if necessary. */
/*options cashost="cloud.example.com" casport=5570;*/
cas casauto;
libname mycas cas;
/* Macro 2 */
%macro myMac2;
    data mycas.class;
       set class;
       y=2;
       put "**** ";
       put " This macro is running in the Compute Server and";
       put " loading a CAS-session table in the Mycas library.";
       put "**** ";
       put _hostname_ 'thread #' _threadid_;
    run;
%mend myMac2;
%myMac2;
```

When this program is executed, in the SAS log, the following text is repeated for each of the 19 observations.

```
****
 This macro is running in the Compute Server and ...
****
host-name thread #1
```

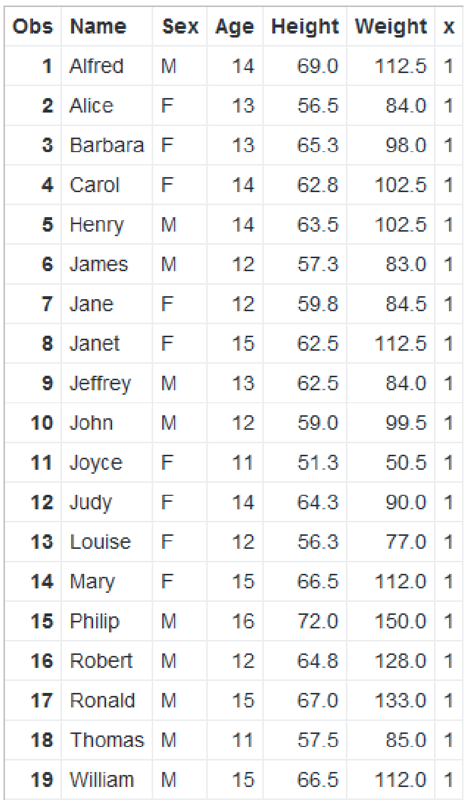
Here is a condensed version of the log.

Example Code 1.1 *Creating the Work.Class Data Set and the Mycas.Class Table*

```
/* Macro 1 */
%macro myMac1;
    data class;
       set sashelp.class;
       x=1;
       put "**** ";
       put " This macro is running in the Compute Server and";
       put " creating a SAS data set in the Work library.";
       put "**** ";
       put _hostname_ 'thread #' _threadid_;
    run;
%mend myMac1;
%myMac1;

****
 This macro is running in the Compute Server and
 creating a SAS data set in the Work library.
****
host-name thread #1
...

NOTE: There were 19 observations read from the data set SASHELP.CLASS.
NOTE: The data set WORK.CLASS has 19 observations and 6 variables.
NOTE: DATA statement used (Total process time):
      real time           0.21 seconds
      cpu time            0.10 seconds



libname mycas cas;
NOTE: Libref MYCAS was successfully assigned as follows:
      Engine:        CAS
      Physical Name: 5f21a217-8baf-6d41-9a46-5ad968ccf5d9
/* Macro 2 */
%macro myMac2;
    data mycas.class;
       set class;
       y=2;
       put "**** ";
       put " This macro is running in the Compute Server and";
       put " loading a CAS-session table in the Mycas library.";
       put "**** ";
       put _hostname_ 'thread #' _threadid_;
    run;
%mend myMac2;
%myMac2;

****
 This macro is running in the Compute Server and
 loading a CAS-session table in the Mycas library.
****
host-name thread #1
...

NOTE: There were 19 observations read from the data set WORK.CLASS.
NOTE: The data set MYCAS.CLASS has 19 observations and 7 variables.
NOTE: DATA statement used (Total process time):
      real time           0.18 seconds
      cpu time            0.12 seconds
```

Here is the Work.Class data set with the X variable added.

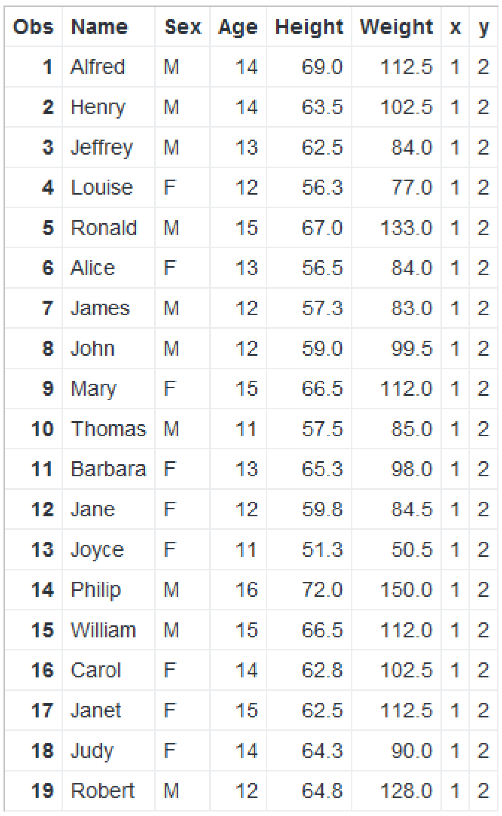
Figure 1.1 *Work.Class Data Set*



Obs	Name	Sex	Age	Height	Weight	x
1	Alfred	M	14	69.0	112.5	1
2	Alice	F	13	56.5	84.0	1
3	Barbara	F	13	65.3	98.0	1
4	Carol	F	14	62.8	102.5	1
5	Henry	M	14	63.5	102.5	1
6	James	M	12	57.3	83.0	1
7	Jane	F	12	59.8	84.5	1
8	Janet	F	15	62.5	112.5	1
9	Jeffrey	M	13	62.5	84.0	1
10	John	M	12	59.0	99.5	1
11	Joyce	F	11	51.3	50.5	1
12	Judy	F	14	64.3	90.0	1
13	Louise	F	12	56.3	77.0	1
14	Mary	F	15	66.5	112.0	1
15	Philip	M	16	72.0	150.0	1
16	Robert	M	12	64.8	128.0	1
17	Ronald	M	15	67.0	133.0	1
18	Thomas	M	11	57.5	85.0	1
19	William	M	15	66.5	112.0	1

Here is the Mycas.Class table with the Y variable added.

Figure 1.2 *Mycas.Class Table*



Obs	Name	Sex	Age	Height	Weight	x	y
1	Alfred	M	14	69.0	112.5	1	2
2	Henry	M	14	63.5	102.5	1	2
3	Jeffrey	M	13	62.5	84.0	1	2
4	Louise	F	12	56.3	77.0	1	2
5	Ronald	M	15	67.0	133.0	1	2
6	Alice	F	13	56.5	84.0	1	2
7	James	M	12	57.3	83.0	1	2
8	John	M	12	59.0	99.5	1	2
9	Mary	F	15	66.5	112.0	1	2
10	Thomas	M	11	57.5	85.0	1	2
11	Barbara	F	13	65.3	98.0	1	2
12	Jane	F	12	59.8	84.5	1	2
13	Joyce	F	11	51.3	50.5	1	2
14	Philip	M	16	72.0	150.0	1	2
15	William	M	15	66.5	112.0	1	2
16	Carol	F	14	62.8	102.5	1	2
17	Janet	F	15	62.5	112.5	1	2
18	Judy	F	14	64.3	90.0	1	2
19	Robert	M	12	64.8	128.0	1	2

Although the macro facility does not exist within CAS, there are several macro variables in the client to help write the code that will run in CAS. For more information, see *SAS Cloud Analytic Services: User's Guide*.

For information about the %MACRO and %MEND statements, see "%MACRO Macro Statement" on page 406 and "%MEND Macro Statement" on page 413.

Replacing Text Strings Using Macro Variables

Macro variables are an efficient way of replacing text strings in SAS code. The simplest way to define a macro variable is to use the %LET statement to assign the macro variable a name (subject to standard SAS naming conventions), and a value.

```
%let city=New Orleans;
```

Now you can use the macro variable CITY in SAS statements where you would like the text `New Orleans` to appear. You refer to the variable by preceding the variable name with an ampersand (&), as in the following TITLE statement:

```
title "Data for &city";
```

The macro processor resolves the reference to the macro variable CITY:

```
   title "Data for New Orleans";
```

A macro variable can be defined within a macro definition or within a statement that is outside a macro definition (called *open code*).

Note: The title is enclosed in double quotation marks. In quoted strings in open code, the macro processor resolves macro variable references within double quotation marks but not within single quotation marks.

A %LET statement in open code (outside a macro definition) creates a global macro variable that is available for use anywhere (except in DATALINES or CARDS statements) in your SAS code during the SAS session in which the variable was created. There are also *local* macro variables, which are available for use only inside the macro definition where they are created. For more information about global and local macro variables, see Scope of Macro Variables on page 57.

Macro variables are not subject to the same length limits as SAS data set variables. The value that you want to assign to a macro variable can contain certain special characters (for example, semicolons, quotation marks, ampersands, and percent signs) or mnemonics (for example, AND, OR, or LT). You must use a macro quoting function to mask the special characters. Otherwise, the special character or mnemonic might be misinterpreted by the macro processor. For more information, see Macro Quoting on page 96.

Macro variables are useful for simple text substitution. They cannot perform conditional operations, DO loops, and other more complex tasks. For this type of work, you must define a macro.

Generating SAS Code Using Macros

Defining Macros

Macros enable you to substitute text in a program and to do many other things. A SAS program can contain any number of macros, and you can invoke a macro any number of times in a single program.

To help you learn how to define your own macros, this section presents a few examples that you can model your own macros after. Each of these examples is fairly simple; by mixing and matching the various techniques, you can create advanced, flexible macros that are capable of performing complex tasks.

Each macro that you define has a distinct name. When choosing a name for your macro, it is recommended that you avoid a name that is a SAS language keyword or call routine name. The name that you choose is subject to the standard SAS naming conventions. A macro name cannot contain double-byte character set (DBCS) characters. A macro definition is placed between a %MACRO statement and a %MEND (macro end) statement, as in the following example:

%MACRO *macro-name*;

%MEND *macro-name*;

The *macro-name* specified in the %MEND statement must match the *macro-name* specified in the %MACRO statement.

Note: Specifying the *macro-name* in the %MEND statement is not required, but it is recommended. It makes matching %MACRO and %MEND statements while debugging easier.

Here is an example of a simple macro definition:

```
%macro dsn;
   Newdata
%mend dsn;
```

This macro is named DSN. `Newdata` is the text of the macro. A string inside a macro is called *constant text* or *model text* because it is the model, or pattern, for the text that becomes part of your SAS program.

To call (or *invoke*) a macro, precede the name of the macro with a percent sign (%):

%*macro-name*

Although the call to the macro looks somewhat like a SAS statement, it does not have to end in a semicolon.

For example, here is how you might call the DSN macro:

```
title "Display of Data Set %dsn";
```

The macro processor executes the macro DSN, which substitutes the constant text in the macro into the TITLE statement:

```
title "Display of Data Set Newdata";
```

Note: The title is enclosed in double quotation marks. In quoted strings in open code, the macro processor resolves macro invocations within double quotation marks but not within single quotation marks.

The macro DSN is exactly the same as the following coding:

```
%let dsn=Newdata;

title "Display of Data Set &dsn";
```

The following code is the result:

```
title "Display of Data Set Newdata";
```

So, in this case, the macro approach does not have any advantages over the macro variable approach. However, DSN is an extremely simple macro. As you will see in later examples, macros can do much more than the macro DSN does.

Inserting Comments in Macros

All code benefits from thorough commenting, and macro code is no exception. There are two types of comments that you can use to add comments to your macro code: PL/1-style comments and macro-style comments. PL/1-style comments begin with `/*` and end with `*/`. PL/1-style comments are commonly used in SAS code. Macro-style comments begin with `%*` and end with a semicolon (`;`). See "%* Macro Comment Macro Statement" on page 382.

The following program uses both types of comments:

```
%macro comment;
/* Here is a PL/1-style comment used in other SAS code. */
   %let myvar=abc;

%* Here is a macro-style comment.;
   %let myvar2=xyz;

%mend comment;
```

You can use whichever type comment that you prefer in your macro code, or use both types as in the previous example.

Asterisk-style comments (**commentary* ;) and COMMENT-style comments (COMMENT *commentary*;) are not recommended within a macro definition. The asterisk-style comment and COMMENT-style comment can be used for constant text. However, any macro statements contained within the comment are executed, which can cause unexpected results. If the comment contains one or more macro statements, two semicolons are required, one to end the macro statement and one to end the comment. If only one semicolon is used, syntax errors or unexpected results can occur. Finally, unmatched quotation marks contained within the comment text are not ignored, which can cause unpredictable results.

TIP PL/1-style comments are the safest comments to use in macro definitions.

For more information about commenting macros, see SAS KB0036213: Using comments within a macro.

Macro Definition Containing Several SAS Statements

You can create macros that contain entire sections of a SAS program:

```
%macro plot;
   proc sgplot;
```

```
      scatter x=height y=weight;
   run;
%mend plot;
```

This macro plots variables HEIGHT and WEIGHT in the last data set that was created. Later in the program that you can invoke the macro:

```
data temp;
   set sashelp.class;
   if age>=12;
run;

%plot;

proc print;
run;
```

When these statements execute, the following program is produced:

```
data temp;
   set sashelp.class;
   if age>=12;
run;

proc sgplot;
   scatter x=height y=weight;
run;

proc print;
run;
```

Passing Information into a Macro Using Parameters

A macro variable defined in parentheses in a %MACRO statement is a *macro parameter*. Macro parameters enable you to pass information into a macro. Here is a simple example:

```
%macro plot(yvar= ,xvar= );
   proc sgplot;
      scatter y=&yvar x=&xvar;
   run;
%mend plot;
```

You invoke the macro by providing values for the parameters:

```
%plot(yvar=weight,xvar=height)

%plot(yvar=weight,xvar=age)
```

When the macro executes, the macro processor matches the values specified in the macro call to the parameters in the macro definition. (This type of parameter is called a *keyword parameter*.)

Macro execution produces the following code:

```
proc sgplot;
   scatter y=weight x=height;
```

```
run;

proc sgplot;
   scatter y=weight x=age;
run;
```

Using parameters has several advantages. First, you can write fewer %LET statements. Second, using parameters ensures that the variables never interfere with parts of your program outside the macro. Macro parameters are an example of *local* macro variables, which exist only during the execution of the macro in which they are defined.

Conditionally Generating SAS Code

By using the %IF-%THEN-%ELSE macro statements, you can conditionally generate SAS code with a macro. See the following example:

```
%macro whatstep(info=, mydata=, yvar=, xvar=);
   %if &info=print %then
      %do;
         proc print data=&mydata;
         run;
      %end;
   %else %if &info=plot %then
      %do;
         ods graphics / reset width=420px height=315px;
         proc sgplot data=&mydata;
            scatter y=&yvar x=&xvar;
         run;
      %end;
%mend whatstep;
```

In this example, the macro WHATSTEP uses keyword parameters, which are set to default null values. When you call a macro that uses keyword parameters, specify the parameter name followed by an equal sign and the value that you want to assign the parameter. Here, the macro WHATSTEP is called with Info set to `print` and MyData set to `sashelp.class`.

```
%whatstep(info=print,mydata=sashelp.class)
```

This code produces the following statements:

```
proc print data=sashelp.class;
run;
```

Because values in the macro processor are case sensitive, the previous program does not work if you specify `PRINT` instead of `print`. To make your macro more robust, use the %UPCASE macro function. For more information, see “%UPCASE Macro Function” on page 371.

Here, the macro WHATSTEP is called with Info set to `plot`, MyData set to `sashelp.class`, yVar set to weight, and xVar set to height:

```
%whatstep(info=plot,mydata=sashelp.class, yvar=weight, xvar=height)
```

This code produces the following statements:

```
ods graphics / reset width=420px height=315px;
proc sgplot data=sashelp.class;
   scatter y=weight x=height;
run;
```

More Advanced Macro Techniques

Generating Repetitive Pieces of Text Using %DO Loops

"Conditionally Generating SAS Code" on page 13 presents a %DO-%END group of statements to conditionally execute several SAS statements. To generate repetitive pieces of text, use an iterative %DO loop. For example, the following macro, NAMES, uses an iterative %DO loop to create a series of names to be used in a DATA statement:

```
%macro names(name= ,number= );
   %do n=1 %to &number;
      &name&n
   %end;
%mend names;
```

The macro NAMES creates a series of names by concatenating the value of the parameter NAME and the value of the macro variable N. You supply the stopping value for N as the value of the parameter NUMBER, as in the following DATA statement:

```
data %names(name=dsn,number=5);
```

Submitting this statement produces the following complete DATA statement:

```
data dsn1 dsn2 dsn3 dsn4 dsn5;
```

Note: You can also execute a %DO loop conditionally with %DO %WHILE and %DO %UNTIL statements. For more information, see "%DO %WHILE Macro Statement" on page 391 and "%DO %UNTIL Macro Statement" on page 390.

Generating a Suffix for a Macro Variable Reference

Suppose that, when you generate a numbered series of names, you always want to put the letter X between the prefix and the number. The macro NAMESX inserts an X after the prefix you supply:

```
%macro namesx(name=,number=);
   %do n=1 %to &number;
      &name.x&n
   %end;
%mend namesx;
```

The period is a delimiter at the end of the reference &NAME. The macro processor uses the delimiter to distinguish the reference &NAME followed by the letter X from the reference &NAMEX. Here is an example of calling the macro NAMESX in a DATA statement:

```
data %namesx(name=dsn,number=3);
```

Submitting this statement produces the following statement:

```
data dsnx1 dsnx2 dsnx3;
```

For more information about using a period as a delimiter in a macro variable reference, see Macro Variables on page 27.

Other Features of the Macro Language

Although subsequent sections go into far more detail on the various elements of the macro language, this section highlights some of the possibilities, with pointers to more information.

macro statements
: This section has illustrated only a few of the macro statements, such as %MACRO and %IF-%THEN. Many other macro statements exist, some of which are valid in open code, and others are valid only in macro definitions. For a complete list of macro statements, see "Macro Statements" on page 190.

macro functions
: Macro functions are functions defined by the macro facility. They process one or more arguments and produce a result. For example, the %SUBSTR function creates a substring of another string, when the %UPCASE function converts characters to uppercase. A special category of macro functions, the macro quoting functions, mask special characters so that they are not misinterpreted by the macro processor.

  There are two special macro functions, %SYSFUNC and %QSYSFUNC, that provide access to SAS language functions or user-written functions generated with SAS/TOOLKIT. You can use %SYSFUNC and %QSYSFUNC with new functions in Base SAS software to obtain the values of SAS host, base, or graphics options. These functions also enable you to open and close SAS data sets, test data set attributes, or read and write to external files. Another special function is %SYSEVALF, which enables your macros to perform floating-point arithmetic.

  For a list of macro functions, see "Macro Functions" on page 192. For a discussion of the macro quoting functions, see Macro Quoting on page 96. For the syntax of calling selected Base SAS functions with %SYSFUNC, see Syntax for Selected Functions with the %SYSFUNC Function on page 531.

autocall macros
: Autocall macros are macros defined by SAS that perform common tasks such as the following:

- trimming leading
- trailing blanks from a macro variable's value
- returning the data type of a value

For a list of autocall macros, see "Selected Autocall Macros Provided with SAS Software" on page 205.

automatic macro variables
: Automatic macro variables are macro variables created by the macro processor. For example, SYSDATE contains the date SAS is invoked. See Chapter 12, "Macro Language Elements," on page 189 for a list of automatic macro variables.

macro facility interfaces
: Interfaces with the macro facility provide a dynamic connection between the macro facility and other parts of SAS, such as the following:

- DATA step
- SCL code
- SQL procedure
- SAS/CONNECT software

For example, you can create macro variables based on values within the DATA step using CALL SYMPUT and retrieve the value of a macro variable stored on a remote host using the %SYSRPUT macro statement. For more information about these interfaces, see Interfaces with the Macro Facility on page 121.

2

SAS Programs and Macro Processing

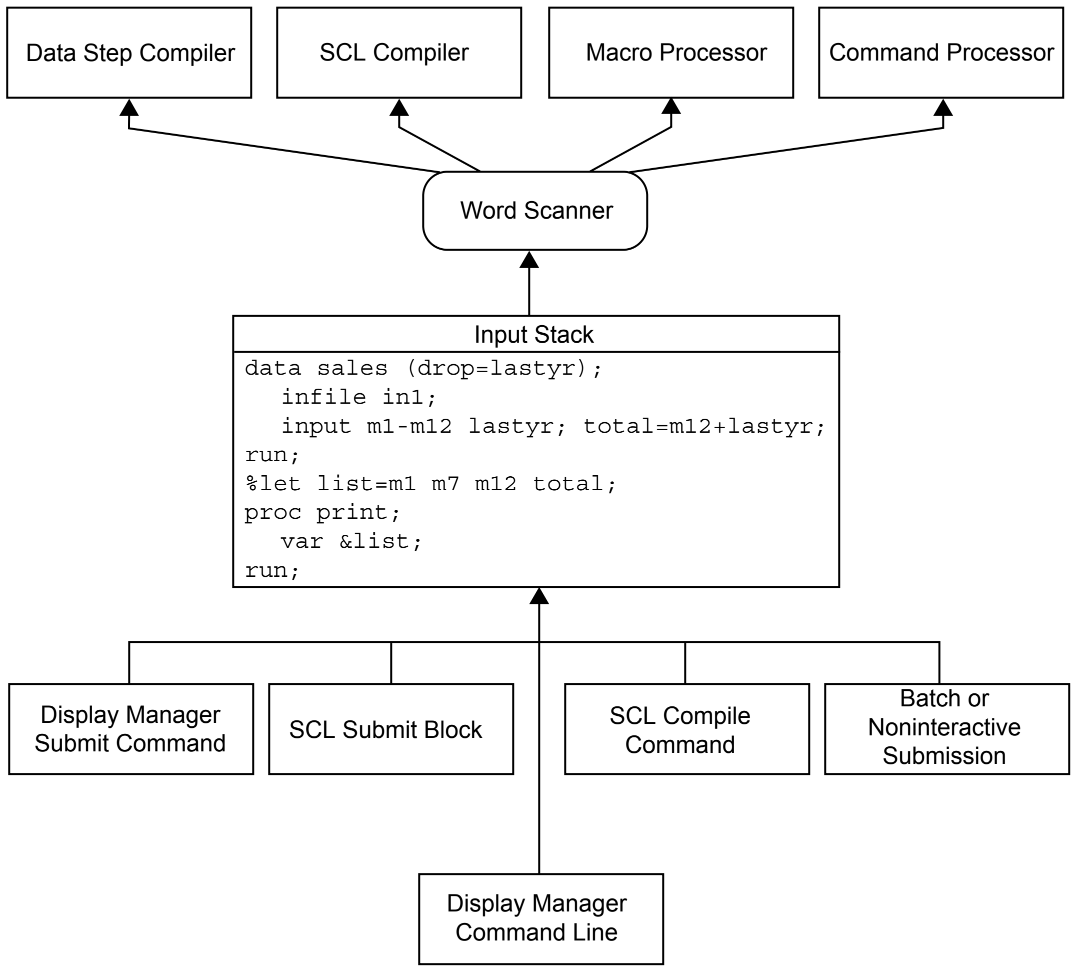
SAS Programs and Macro Processing

This section describes the typical pattern that SAS follows to process a program. These concepts are helpful for understanding how the macro processor works with other parts of SAS. However, they are not required for most macro programming. They are provided so that you can understand what is going on behind the scenes.

Note: The concepts in this section present a logical representation, not a detailed physical representation, of how SAS software works.

When you submit a program, it goes to an area of memory called the *input stack*. This is true for all program and command sources: the SAS windowing environment, the SCL SUBMIT block, the SCL COMPILE command, or from batch or noninteractive sessions. The input stack shown in the following figure contains a simple SAS program that displays sales data. The first line in the program is the top of the input stack.

Figure 2.1 Submitted Programs Are Sent to the Input Stack



```
data sales (drop=lastyr);
   infile in1;
   input m1-m12 lastyr; total=m12+lastyr;
run;
%let list=m1 m7 m12 total;
proc print;
   var &list;
run;
```

Once a program reaches the input stack, SAS transforms the stream of characters into individual tokens. These tokens are transferred to different parts of SAS for processing, such as the DATA step compiler and the macro processor. Knowing how SAS recognizes tokens and how they are transferred to different parts of SAS will help you understand how the various parts of SAS and the macro processor work together. Also, how to control the timing of macro execution in your programs. The following sections show you how a simple program is tokenized and processed.

How SAS Processes Statements without Macro Activity

The process that SAS uses to extract words and symbols from the input stack is called *tokenization*. Tokenization is performed by a component of SAS called the *word scanner*, as shown in Figure 2.2 on page 19. The word scanner starts at the first character in the input stack and examines each character in turn. In doing so, the word scanner assembles the characters into tokens. There are four general types of tokens:

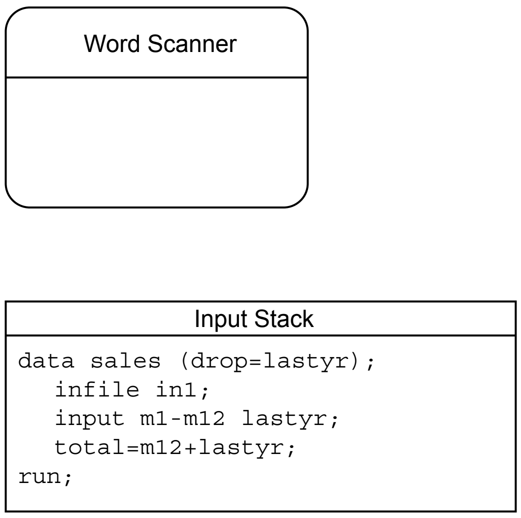
Literal
: a string of characters enclosed in quotation marks.

Number
: digits, date values, time values, and hexadecimal numbers.

Name
: a string of characters beginning with an underscore or letter.

Special
: any character or group of characters that have special meaning to SAS. Examples of special characters include: `* / + - ** ; $ ( ) . & % =`

Figure 2.2 The Sample Program before Tokenization



```
data sales (drop=lastyr);
   infile in1;
   input m1-m12 lastyr;
   total=m12+lastyr;
run;
```

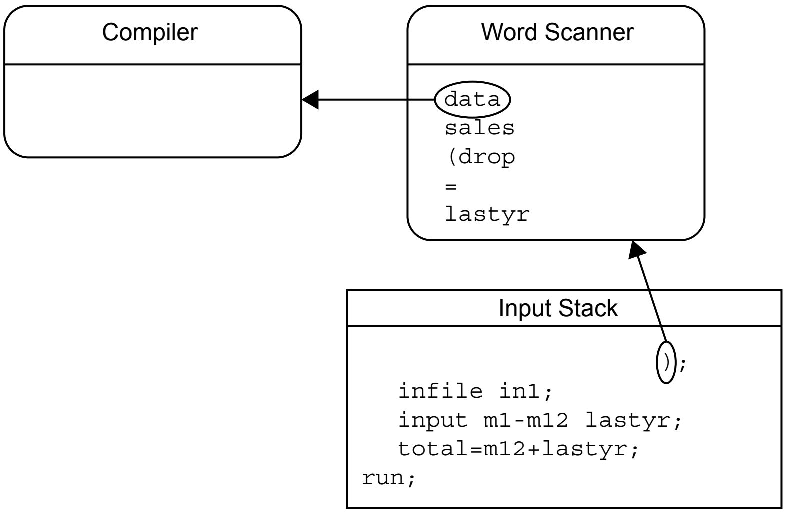
The first SAS statement in the input stack in the preceding figure contains eight tokens (four names and four special characters).

```
data sales(drop=lastyr);
```

When the word scanner finds a blank or the beginning of a new token, it removes a token from the input stack and transfers it to the bottom of the queue.

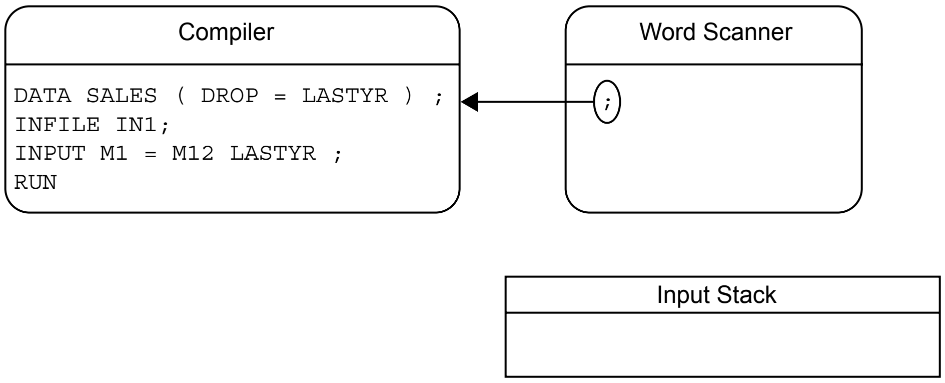
In this example, when the word scanner pulls the first token from the input stack, it recognizes the token as the beginning of a DATA step. The word scanner triggers the DATA step compiler, which begins to request more tokens. The compiler pulls tokens from the top of the queue, as shown in the following figure.

Figure 2.3 The Word Scanner Obtains Tokens



The compiler continues to pull tokens until it recognizes the end of the DATA step (in this case, the RUN statement), which is called a DATA step boundary, as shown in the following figure. When the DATA step compiler recognizes the end of a step, the step is executed, and the DATA step is complete.

Figure 2.4 The Word Scanner Sends Tokens to the Compiler

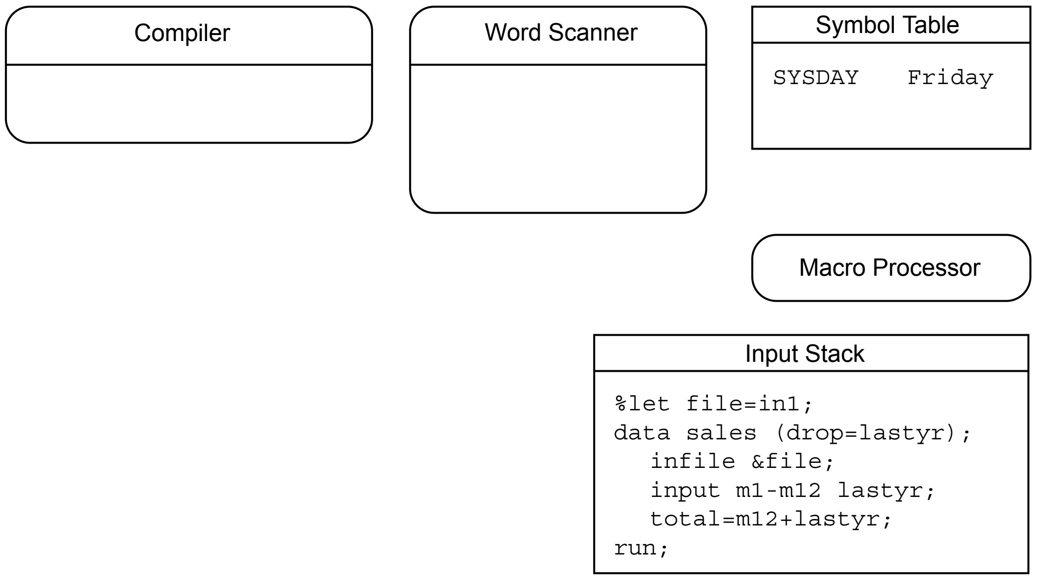


In most SAS programs with no macro processor activity, all information that the compiler receives comes from the submitted program.

How SAS Processes Statements with Macro Activity

In a program with macro activity, the macro processor can generate text that is placed on the input stack to be tokenized by the word scanner. The example in this section shows you how the macro processor creates and resolves a macro variable. To illustrate how the compiler and the macro processor work together, the following figure contains the macro processor and the macro variable symbol table. SAS creates the symbol table to hold the values of automatic and global macro variables. SAS creates automatic macro variables at the beginning of a Compute Server session. For the sake of illustration, the symbol table is shown with only one automatic macro variable, SYSDAY.

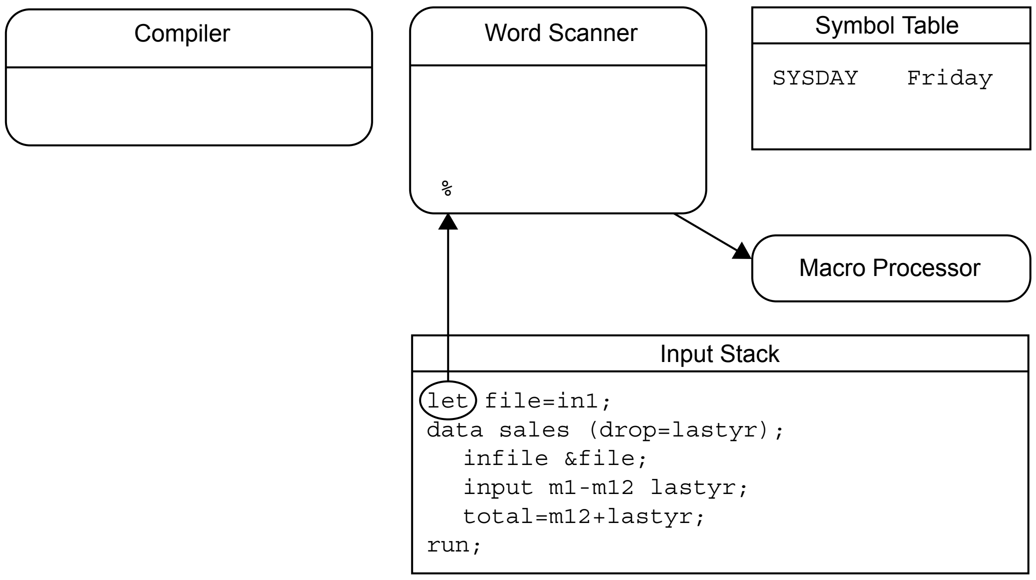
Figure 2.5 *The Macro Processor and Symbol Table*



```
%let file=in1;
data sales (drop=lastyr);
   infile &file;
   input m1-m12 lastyr;
   total=m12+lastyr;
run;
```

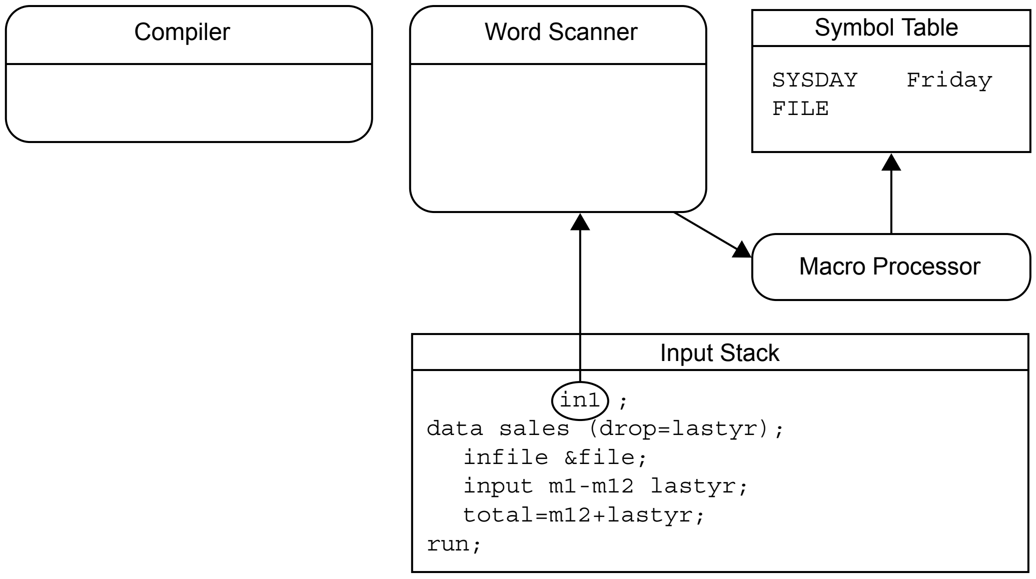
Whenever the word scanner encounters a macro trigger, it sends information to the macro processor. A macro trigger is either an ampersand (&) or percent sign (%) followed by a nonblank character. As it did in the previous example, the word scanner begins to process this program by examining the first characters in the input stack. In this case, the word scanner finds a percent sign (%) followed by a nonblank character. The word scanner recognizes this combination of characters as a potential macro language element, and triggers the macro processor to examine % and LET, as shown in the following figure.

Figure 2.6 The Macro Processor Examines LET



When the macro processor recognizes a macro language element, it begins to work with the word scanner. In this case, the macro processor removes the %LET statement, and writes an entry in the symbol table, as shown in the following figure.

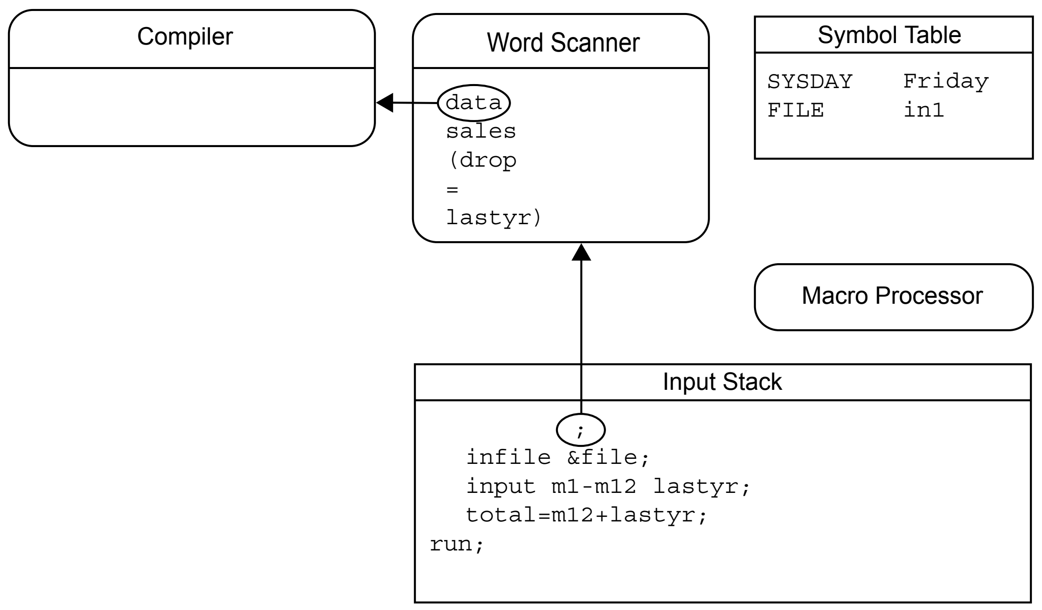
Figure 2.7 The Macro Processor Writes to the Symbol Table



From the time the word scanner triggers the macro processor until that macro processor action is complete, the macro processor controls all activity. When the macro processor is active, no activity occurs in the word scanner or the DATA step compiler.

When the macro processor is finished, the word scanner reads the next token (the DATA keyword in this example) and sends it to the compiler. The word scanner triggers the compiler, which begins to pull tokens from the top of the queue, as shown in the following figure.

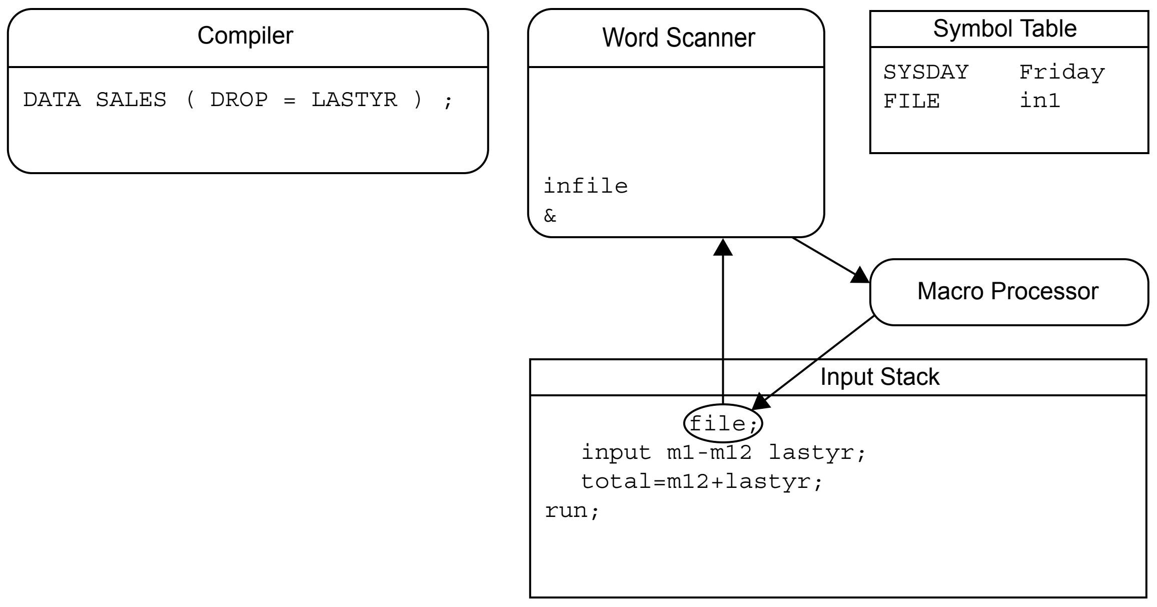
Figure 2.8 *The Word Scanner Resumes Tokenization*



As it processes each token, SAS removes the protection that the macro quoting functions provide to mask special characters and mnemonic operators. For more information, see Chapter 2, “SAS Programs and Macro Processing,” on page 17.

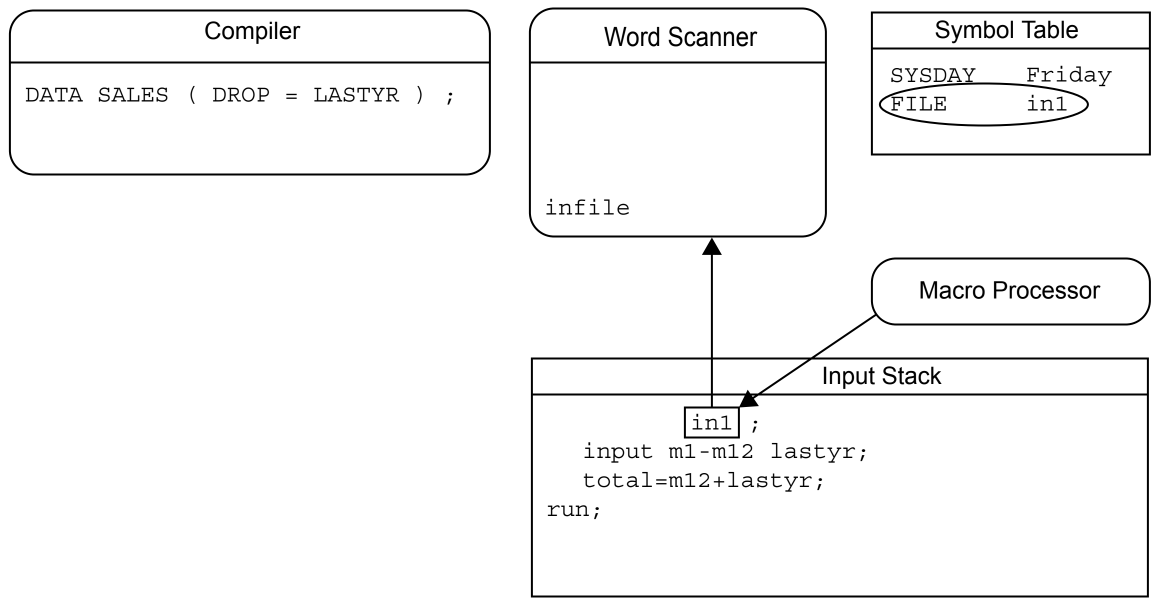
If the word scanner finds an ampersand followed by a nonblank character in a token, it triggers the macro processor to examine the next token, as shown in the following figure.

Figure 2.9 *The Macro Processor Examines &FILE*



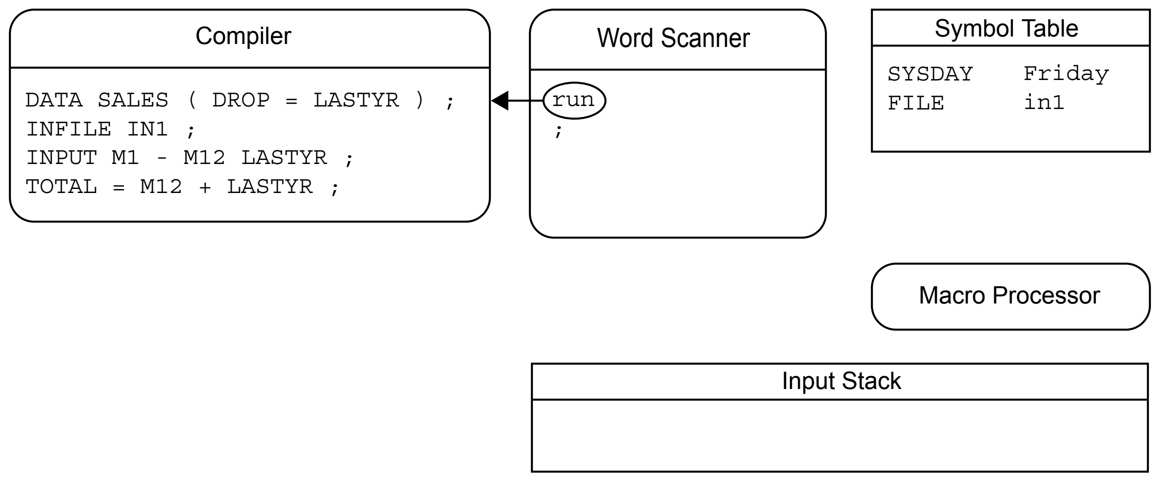
The macro processor examines the token and recognizes a macro variable that exists in the symbol table. The macro processor removes the macro variable name from the input stack and replaces it with the text from the symbol table, as shown in the following figure.

Figure 2.10 The Macro Processor Generates Text to the Input Stack



The compiler continues to request tokens, and the word scanner continues to supply them, until the entire input stack has been read as shown in the following figure.

Figure 2.11 The Word Scanner Completes Processing



If the end of the input stack is a DATA step boundary, as it is in this example, the compiler compiles and executes the step. SAS then frees the DATA step task. Any macro variables that were created during the program remain in the symbol table. If the end of the input stack is not a step boundary, the processed statements remain in the compiler. Processing resumes when more statements are submitted to the input stack.

3

Macro Variables

Macro Variables

Macro variables are tools that enable you to dynamically modify the text in a SAS program through symbolic substitution. You can assign large or small amounts of text to macro variables, and after that, you can use that text by simply referencing the variable that contains it.

Macro variable values have a maximum length of 65,534 bytes. The length of a macro variable is determined by the text assigned to it instead of a specific length declaration. So its length varies with each value that it contains. Macro variables contain only character data. However, the macro facility has features that enable a variable to be evaluated as a number when it contains character data that can be interpreted as a number. The value of a macro variable remains constant until it is specifically changed. Macro variables are independent of SAS data set variables.

Note: Only printable characters should be assigned to macro variables. Non-printable values that are assigned to macro variables can cause unpredictable results.

Macro variables defined by macro programmers are called *user-defined macro variables*. Those defined by the macro processor are called *automatic macro variables*. You can define and use macro variables anywhere in SAS programs, except within data lines.

When a macro variable is defined, the macro processor adds it to one of the program's macro variable symbol tables. The variable is held in the global symbol table, which the macro processor creates at the beginning of a Compute Server session when the following occurs:

- a macro variable is defined in a statement that is outside a macro definition (called *open code*)
- the variable is created automatically by the macro processor (except SYSPBUFF)

When a macro variable is defined within a macro and is not specifically defined as global, the variable is typically held in the macro's local symbol table. SAS creates the local symbol table when the macro starts executing. For more information about symbol tables, see Chapter 2, "SAS Programs and Macro Processing," on page 17 and Chapter 5, "Scopes of Macro Variables," on page 57.

When it is in the global symbol table, a macro variable exists for the remainder of the current Compute Server session. A variable in the global symbol table is called a *global macro variable*. This variable has global scope because its value is available to any part of the session (except in CARDS or DATALINES statements). Other components of SAS might create global macro variables, but only those components created by the macro processor are considered *automatic macro variables*.

When it is in a local symbol table, a macro variable exists only during execution of the macro in which it is defined. A variable in a local symbol table is called a *local macro variable*. It has local scope because its value is available only while the macro is executing. Chapter 5, "Scopes of Macro Variables," on page 57 contains figures that illustrate a program with a global and a local symbol table.

You can use the %PUT statement to view all macro variables available in a current Compute Server session. For more information, see "%PUT Macro Statement" on page 414 and also Chapter 10, "Macro Facility Error Messages and Debugging," on page 145.

Macro Variables Defined by the Macro Processor

When you invoke SAS, the macro processor creates automatic macro variables that supply information related to the Compute Server session. Automatic variables are global except SYSPBUFF, which is local.

To use an automatic macro variable, reference it with an ampersand followed by the macro variable name (for example, &SYSJOBID). This FOOTNOTE statement contains references to the automatic macro variables SYSDAY and SYSDATE9:

```
footnote "Report for &sysday, &sysdate9";
```

If the current Compute Server session is invoked on December 16, 2011, macro variable resolution causes SAS to receive this statement:

```
FOOTNOTE "Report for Friday, 16DEC2011";
```

Automatic macro variables are often useful in conditional logic such as a %IF statement with actions determined by the value that is returned. For more information, see "%IF-%THEN/%ELSE Macro Statement" on page 397.

You can assign values to automatic macro variables that have read and write status. However, you cannot assign a value to an automatic macro variable that has read-only status. The following table lists the automatic macro variables that are created by the SAS macro processor and the read and write status.

Use %PUT _AUTOMATIC_ to view all available automatic macro variables:

```
   %put _automatic_;
```

The automatic macro variables are written to the SAS log.

There are also system-specific macro variables that are created only on a particular platform. These are documented in the host companion, and common ones are listed in "Writing Efficient and Portable Macros" on page 171. Other SAS software products also provide macro variables, which are described in the documentation for the product that uses them. Neither of these types of macro variables are considered automatic macro variables.

Table 3.1 Automatic Macro Variables by Category

Status	Variable	Contains
	SYSCC	The current condition code that SAS returns to your operating environment (the operating environment condition code)
	SYSDEVIC	Name of current graphics device

Status	Variable	Contains
	SYSDMG	Return code that reflects an action taken on a damaged data set
	SYSDSN	Name of most recent SAS data set in two fields
	SYSFILRC	Return code set by the FILENAME statement
	SYSLAST	Name of most recent SAS data set in one field
	SYSLCKRC	Return code set by the LOCK statement
	SYSLIBRC	Return code set by the LIBNAME statement
	SYSLOGAPPLNAME	Value of the LOGAPPLNAME option
	SYSPARM	Value specified with the SYSPARM= system option
	SYSPBUFF	Text of macro parameter values
	SYSRC	Various system-related return codes
Read-only	SYSADDRBITS	The number of bits of an address
	SYSCHARWIDTH	The character width value
	SYSDATASTEPPHASE	Value of the current running phase of the DATA step
	SYSDATE	The character value representing the date on which a SAS job or session began executing (two-digit year)
	SYSDATE9	The character value representing the date on which a SAS job or session began executing (four-digit year)
	SYSDAY	Day of week on which SAS job or session began executing
	SYSENCODING	Name of the Compute Server session encoding
	SYSENDIAN	An indication of the byte order of the current session

Status	Variable	Contains
	SYSENV	Foreground or background indicator
	SYSERR	Return code set by SAS procedures and the DATA step
	SYSERRORTEXT	Text of the last error message formatted for display on the SAS log
	SYSHOSTINFOLONG	The operating environment information if the HOSTINFOLONG option is specified
	SYSHOSTNAME	The host name of the operating environment
	SYSINCLUDEFILEDEVICE	The device type of the current %INCLUDE file.
	SYSINCLUDEFILEDIR	The directory where the current %INCLUDE file was found.
	SYSINCLUDEFILEFILEREF	The fileref associated with the current %INCLUDE file or blank.
	SYSINCLUDEFILENAME	The filename of the current %INCLUDE file.
	SYSINDEX	Number of macros that have begun execution during this session
	SYSINFO	Return code information
	SYSJOBID	Name of current batch job or user ID (varies by host environment)
	SYSMACRONAME	Name of current executing macro
	SYSMAXLONG	Returns the maximum long integer value allowed under Linux.
	SYSMENV	Current macro execution environment
	SYSNCPU	The current number of processors that SAS might use in computation
	SYSNOBS	The number of observations in the last data set created by a procedure or DATA step

Status	Variable	Contains
	SYSODSESCAPECHAR	The value of the ODS ESCAPECHAR= from within the program
	SYSODSPATH	The value of the PATH variable in the Output Delivery System (ODS)
	SYSPRINTTOLIST	The path of the LIST file set by the PRINTTO procedure for future redirection
	SYSPRINTTOLOG	The path of the LOG file set by the PRINTTO procedure for future redirection
	SYSPROCESSID	The process ID of the current SAS process
	SYSPROCESSMODE	The name of the current Compute Server session run mode or server type
	SYSPROCESSNAME	The process name of the current SAS process
	SYSPROCNAME	The name of current procedure being processed
	SYSSCP	The abbreviation of an operating system
	SYSSCPL	The name of an operating system
	SYSSITE	The number assigned to your site
	SYSSIZEOFLONG	The length in bytes of a long integer in the current session
	SYSSIZEOFPTR	The size in bytes of a pointer
	SYSSIZEOFUNICODE	The length in bytes of a Unicode character in the current session
	SYSSTARTID	The ID generated from the last STARTSAS statement
	SYSSTARTNAME	The process name generated from the last STARTSAS statement
	SYSTCPIPHOSTNAME	The host names of the local and remote operating environments when multiple TCP/IP stacks are supported

Status	Variable	Contains
	SYSTIME	The character value of the time at which a SAS job or session began executing
	SYSTIMEZONE	The time zone name based on TIMEZONE option
	SYSTIMEZONEIDENT	The time zone ID based on TIMEZONE option
	SYSTIMEZONEOFFSET	The current time zone offset based on TIMEZONE option
	SYSUSERID	The user ID or login of the current SAS process
	SYSVER	The release or version number of SAS software executing
	SYSVIYARELEASE	The cadence release number of the SAS Viya platform software
	SYSVIYAVERSION	The cadence version of the SAS Viya platform software
	SYSVLONG	The release number and maintenance level of SAS software with a 2-digit year
	SYSVLONG4	The release number and maintenance level of SAS software with a 4-digit year
	SYSWARNINGTEXT	Text of the last warning message formatted for display on the SAS log

Macro Variables Defined by Users

Overview for Defining Macro Variables

You can create your own macro variables, change their values, and define their scope. You can define a macro variable within a macro, and you can also specifically define it as a global variable, by defining it with the %GLOBAL statement. Macro variable names must start with a letter or an underscore and can be followed by letters or digits. You can assign any name to a macro variable as long as the name is not a reserved word. The prefixes AF, DMS, SQL, and SYS are not recommended because they are frequently used in SAS software when creating macro variables. Thus, using one of these prefixes can cause a name conflict with macro variables created by SAS software. For a complete list of reserved words in the macro language, see Appendix 1, “ Reserved Words in the Macro Facility,” on page 483. If you assign a macro variable name that is not valid, an error message is printed in the SAS log.

You can use %PUT _ALL_ to view all user-created macro variables.

Creating User-Defined Macro Variable Names

The simplest way to create a user-defined macro variable is to use the macro program statement %LET:

```
%let dsname=Newdata;
```

DSNAME is the name of the macro variable. `Newdata` is the value of the macro variable DSNAME. The following are the rules for creating a macro variable:

1 SAS macro variable names can be up to 32 characters in length.

2 The first character must begin with a letter or an underscore. Subsequent characters can be letters, numeric digits, or underscores.

3 A macro variable name cannot contain blanks.

4 A macro variable name cannot contain double-byte character set (DBCS) characters.

5 A macro variable name cannot contain any special characters other than the underscore.

6 Macro variable names are case insensitive. For example, cat, Cat, and CAT all represent the same variable.

7 You can assign any name to a macro variable as long as the name is not a reserved word. The prefixes AF, DMS, SQL, and SYS are not recommended because they are frequently used in SAS software for automatic macro variables. Thus, using one of these prefixes can cause a name conflict with an automatic macro variable. For a complete list of reserved words in the macro language, see Appendix 1, “ Reserved Words in the Macro Facility,” on page 483. If you assign a macro variable name that is not valid, an error message is printed in the SAS log.

Assigning Values to Macro Variables

The simplest way to assign a value to a macro variable is to use the macro program statement %LET:

```
%let dsname=Newdata;
```

DSNAME is the name of the macro variable. `Newdata` is the value of the macro variable DSNAME. The value of a macro variable is simply a string of characters. The characters can include any letters, numbers, or printable symbols found on your keyboard, and blanks between characters. The case of letters is preserved in a macro variable value. Some characters, such as unmatched quotation marks, require special treatment, which is described later.

If a macro variable already exists, a value assigned to it replaces its current value. If a macro variable or its value contains macro triggers (% or &), the trigger is evaluated before the value is assigned. In the following example, `&name` is resolved to `Cary` and then it is assigned as the value of `city` in the following statements:

```
%let name=Cary;
%let city=&name;
```

Generally, the macro processor treats alphabetic characters, digits, and symbols (except & and %) as characters. It can also treat & and % as characters using a special treatment, which is described later. It does not make a distinction between character and numeric values as the rest of SAS does. (However, the “%EVAL Macro Function” on page 316 and “%SYSEVALF Macro Function” on page 352 can evaluate macro variables as integers or floating point numbers.)

Macro variable values can represent text to be generated by the macro processor or text to be used by the macro processor. Values can range in length from 0 to 65,534 characters. If you omit the value argument, the value is null (0 characters). By default, leading and trailing blanks are not stored with the value.

In addition to the %LET statement, the following list contains other features of the macro language that create macro variables:

- iterative %DO statement
- %GLOBAL statement
- INTO clause of the SELECT statement in SQL
- %LOCAL statement
- %MACRO statement
- SYMPUT and SYMPUTX routine and SYMPUTN routine in SCL

The following table describes how to assign a variety of types of values to macro variables.

Table 3.2 Types of Assignments for Macro Variable Values

Assign	Values
Constant text	A character string. The following statements show several ways that the value `maple` can be assigned to macro variable STREET. In each case, the macro processor stores the five-character value `maple` as the value of STREET. The leading and trailing blanks are not stored. `%let street=maple;`

Assign	Values

	`%let street=            maple;`
	`%let street=maple            ;`
	Note: Quotation marks are not required. If quotation marks are used, they become part of the value.

Digits	The appropriate digits. This example creates the macro variables NUM and TOTALSTR:
	`%let num=123;`
	`%let totalstr=100+200;`
	The macro processor does not treat `123` as a number or evaluate the expression `100+200`. Instead, the macro processor treats all the digits as characters.

Arithmetic expressions	The %EVAL function, for example,
	`%let num=%eval(100+200); / * produces 300 * /`
	use the %SYSEVALF function, for example,
	`%let num=%sysevalf(100+1.597); / * produces 101.597 * /`
	For more information, see "Macro Evaluation Functions" on page 195.

A null value	No assignment for the value argument, for example, `%let country=;`

A macro variable reference	A macro variable reference, *&macro-variable*. For example,
	`%let street=Maple;`
	`%let num=123;`
	`%let address=&num &street Avenue;`
	This example shows multiple macro references that are part of a text expression. The macro processor attempts to resolve text expressions before it makes the assignment. Thus, the macro processor stores the value of macro variable ADDRESS as `123 Maple Avenue`.
	You can treat ampersands and percent signs as literals by using the %NRSTR function to mask the character. This causes the macro processor to treat it as text instead of trying to interpret it as a macro call. For more information, see Chapter 12, "Macro Language Elements," on page 189 and Macro Quoting on page 96.

A macro invocation	A macro call, *%macro-name*. For example, `%let status=%wait;` When the %LET statement executes, the macro processor also invokes the macro WAIT. The macro processor stores the text produced by the macro WAIT as the value of STATUS. To prevent the macro from being invoked when the %LET statement executes, use the %NRSTR function to mask the percent sign:

Assign	Values
	`%let status=%nrstr(%wait);` The macro processor stores `%wait` as the value of STATUS.

Blanks and special characters	Macro quoting function %STR or %NRSTR around the value. This action masks the blanks or special characters so that the macro processor interprets them as text. For more information, see “Macro Quoting Functions” on page 196. For example,
	
	`%let state=%str( North Carolina);`
	`%let town=%str(Taylor%'s Pond);`
	`%let store=%nrstr(Smith&Jones);`
	`%let plotit=%str(`
	`       proc sgplot;`
	`         scatter y=weight x=height;`
	`       run;);`
	
	The definition of macro variable TOWN demonstrates using %STR to mask a value containing an unmatched quotation mark. “Macro Quoting Functions” on page 196 discuss macro quoting functions that require unmatched quotation marks and other symbols to be marked.
	
	The definition of macro variable PLOTIT demonstrates using %STR to mask blanks and special characters (semicolons) in macro variable values. When a macro variable contains complete SAS statements, the statements are easier to read if you enter them on separate lines with indentions for statements within a DATA or PROC step. Using a macro quoting function retains the significant blanks in the macro variable value.

Value from a DATA step	The SYMPUT routine. This example puts the number of observations in a data set into a FOOTNOTE statement where AGE is greater than 12:
	
	`data _null_;`
	`   set sashelp.class end=final;`
	`   if age>12 then n+1;`
	`   if final then call symput('number',strip(put(n, 5.)));`
	`run;`
	`footnote "&number Observations have AGE>12";`
	
	During the last iteration of the DATA step, the SYMPUT routine creates a macro variable named NUMBER whose value is the value of N. The PUT function converts the DATA step variable N to a character string and the STRIP function removes the leading blanks before the value is assigned to the macro variable NUMBER.
	
	For a discussion of SYMPUT, see “CALL SYMPUT Routine” on page 289.

Using Macro Variables

Macro Variable Reference

After a macro variable is created, you typically use the variable by referencing it with an ampersand preceding its name (&*variable-name*), which is called a *macro variable reference*. These references perform symbolic substitutions when they resolve to their value. You can use these references anywhere in a SAS program. To resolve a macro variable reference that occurs within a literal string, enclose the string in double quotation marks. Macro variable references that are enclosed in single quotation marks are not resolved. Compare the following statements that assign a value to macro variable DSN and use it in a TITLE statement:

```
%let dsn=sashelp.class;
title1 "Contents of Data Set &dsn";
title2 'Contents of Data Set &dsn';
```

In the first TITLE statement, the macro processor resolves the reference by replacing &DSN with the value of macro variable DSN. In the second TITLE statement, the value for DSN does not replace &DSN. SAS sees the following statements:

```
TITLE1 "Contents of Data Set sashelp.class";
TITLE2 'Contents of Data Set &dsn';
```

You can refer to a macro variable as many times as you need to in a SAS program. The value remains constant until you change it. For example, this program refers to macro variable DSN twice:

```
%let dsn=sashelp.class;
data temp;
   set &dsn;
   if age>=12;
run;

proc print;
   title "Subset of Data Set &dsn";
run;
```

Each time the reference &DSN appears, the macro processor replaces it with `sashelp.class`. SAS sees the following statements:

```
DATA TEMP;
   SET SASHELP.CLASS;
   IF AGE>=12;
RUN;

PROC PRINT;
   TITLE "Subset of Data Set sashelp.class";
```

```
RUN;
```

Note: If you reference a macro variable that does not exist, a warning message is printed in the SAS log. For example, if macro variable JERRY is misspelled as JERY, the following produces an unexpected result:

```
%let jerry=student;
data temp;
  x="produced by &jery";
run;
```

This code produces the following message:

```
WARNING:  Apparent symbolic reference JERY not resolved.
```

Combining Macro Variable References with Text

It is often useful to place a macro variable reference next to leading or trailing text (for example, DATA=PERSNL&YR.EMPLOYES, where &YR contains two characters for a year), or to reference adjacent variables (for example, &MONTH&YR). You can reuse the same text in several places or to reuse a program because you can change values for each use.

To reuse the same text in several places, you can write a program with macro variable references representing the common elements. You can change all the locations with a single %LET statement, as shown:

```
%let name=class;
data new&name;
   set sashelp.&name;
   more SAS statements
   if age>12;
run;
```

After macro variable resolution, SAS sees these statements:

```
DATA NEWCLASS;
   SET SASHELP.CLASS;
   more SAS statements
   IF age>12;
   RUN;
```

Notice that macro variable references do not require the concatenation operator as the DATA step does. SAS forms the resulting words automatically.

Delimiting Macro Variable Names within Text

Sometimes when you use a macro variable reference as a prefix, the reference does not resolve as you expect if you simply concatenate it. Instead, you might need to delimit the reference by adding a period to the end of it.

A period immediately following a macro variable reference acts as a delimiter. That is, a period at the end of a reference forces the macro processor to recognize the end of the reference. The period does not appear in the resulting text.

Continuing with the example above, suppose that you need another DATA step that uses the names Sales1, Sales2, and Insales.Temp. You might add the following step to the program:

```
/*  first attempt to add suffixes--incorrect  */
%let name = sales;
data &name1 &name2;
   set in&name.temp;
run;
```

After macro variable resolution, SAS sees these statements:

```
DATA &NAME1 &NAME2;
   SET INSALESTEMP;
RUN;
```

None of the macro variable references have resolved as you intended. The macro processor issues warning messages, and SAS issues syntax error messages. Why?

Because NAME1 and NAME2 are valid SAS names, the macro processor searches for those macro variables rather than for NAME, and the references pass into the DATA statement without resolution.

In a macro variable reference, the word scanner recognizes that a macro variable name has ended when it encounters a character that is not used in a SAS name. However, you can use a period (.) as a delimiter for a macro variable reference. For example, to cause the macro processor to recognize the end of the word NAME in this example, use a period as a delimiter between &NAME and the suffix:

```
/*  correct version  */
data &name.1 &name.2;
```

SAS now sees this statement:

```
DATA SALES1 SALES2;
```

Creating a Period to Follow Resolved Text

Sometimes you need a period to follow the text resolved by the macro processor. For example, a two-level data set name needs to include a period between the libref and data set name.

When the character following a macro variable reference is a period, use two periods. The first is the delimiter for the macro reference, and the second is part of the text.

```
set in&name..temp;
```

After macro variable resolution, SAS sees this statement:

```
SET INSALES.TEMP;
```

You can end any macro variable reference with a delimiter, but the delimiter is necessary only if the characters that follow can be part of a SAS name. For example, both of these TITLE statements are correct:

```
title "&name.--a report";
title "&name--a report";
```

They produce the following:

```
TITLE "sales--a report";
```

Displaying Macro Variable Values

The simplest way to display macro variable values is to use the %PUT statement, which writes text to the SAS log. For example, the following statements write the following result:

```
%let a=first;
%let b=macro variable;
%put &a ***&b***;
```

Here is the result:

```
first ***macro variable***
```

You can also use a "%PUT Macro Statement" on page 414 to view available macro variables. %PUT provides several options that enable you to view individual categories of macro variables.

The system option SYMBOLGEN displays the resolution of macro variables. For this example, assume that macro variables PROC and DSET have the values SGPLOT and Sasuser.Houses, respectively.

```
options symbolgen;
title "%upcase(&proc) of %upcase(&dset)";
options nosymbolgen;
```

The SYMBOLGEN option prints to the log:

```
SYMBOLGEN:  Macro variable PROC resolves to sgplot
SYMBOLGEN:  Macro variable DSET resolves to sasuser.houses
```

The NOSYMBOLGEN option disables the display of macro variable resolution.

For more information about debugging macro programs, see Chapter 10, "Macro Facility Error Messages and Debugging," on page 145.

Referencing Macro Variables Indirectly

Using an Expression to Generate a Reference

The macro variable references shown so far have been direct macro references that begin with one ampersand: &*name*. However, it is also useful to be able to indirectly reference macro variables that belong to a series so that the name is determined when the macro variable reference resolves. The macro facility provides indirect macro variable referencing, which enables you to use an expression (for example, CITY&N) to generate a reference to one of a series of macro variables. For example, you could use the value of macro variable N to reference a variable in the series of macro variables named CITY1 to CITY20. If N has the value 8, the reference would be to CITY8. If the value of N is 3, the reference would be to CITY3.

Although for this example the type of reference that you want is CITY&N, the following example will not produce the value of &N appended to CITY:

```
%put &city&n;  /* incorrect */
```

This code produces a warning message saying that there is no macro variable CITY because the macro facility has tried to resolve &CITY and then &N and concatenate those values.

When you use an indirect macro variable reference, you must force the macro processor to scan the macro variable reference more than once. This process will resolve the desired reference on the second, or later, scan. To force the macro processor to rescan a macro variable reference, you use more than one ampersand in the macro variable reference. When the macro processor encounters multiple ampersands, its basic action is to resolve two ampersands to one ampersand. For example, for you to append the value of &N to CITY and then reference the appropriate variable name, do the following:

```
%put &&city&n;  /* correct */
```

If &N contains 6, when the macro processor receives this statement, it performs the following steps:

1 resolves && to &

2 passes CITY as text

3 resolves &N into 6

4 returns to the beginning of the macro variable reference, &CITY6, starts resolving from the beginning again, and prints the value of CITY6

Generating a Series of Macro Variable References with a Single Macro Call

Using indirect macro variable references, you can generate a series of references with a single macro call by using an iterative %DO loop:

```
%let city1=Cary;
%let city2=New York;
%let city3=Chicago;
%let city4=Los Angeles;
%let city5=Dallas;

%macro listthem;
   %do n=1 %to 5;
      &&city&n
   %end;
%mend listthem;

%put %listthem;
```

This program writes the following to the SAS log:

```
Cary        New York        Chicago        Los Angeles        Dallas
```

Using More Than Two Ampersands

You can use any number of ampersands in an indirect macro variable reference, although using more than three is rare. Regardless of how many ampersands are used in this type of reference, the macro processor performs the following steps to resolve the reference.

```
%let var=city;
%let n=6;
%put &&&var&n;
```

1 It resolves the entire reference from left-to-right. If a pair of ampersands (&&) is encountered, the pair is resolved to a single ampersand, then the next part of the reference is processed. In this example, &&&VAR&N becomes &CITY6.

2 It returns to the beginning of the preliminary result and starts resolving again from left-to-right. When all ampersands have been fully processed, the resolution is complete. In this example, &CITY6 resolves to Boston, and the resolution process is finished.

Note: A macro call cannot be part of the resolution during indirect macro variable referencing.

TIP In some cases, using indirect macro references with triple ampersands increases the efficiency of the macro processor. For more information, see Chapter 11, "Writing Efficient and Portable Macros," on page 171.

Manipulating Macro Variable Values with Macro Functions

When you define macro variables, you can include macro functions in the expressions to manipulate the value of the variable before the value is stored. For example, you can use functions that scan other values, evaluate arithmetic and logical expressions, and remove the significance of special characters such as unmatched quotation marks.

To scan for words in macro variable values, use the %SCAN function:

```
%let address=123 maple avenue;
%let frstword=%scan(&address,1);
```

The first %LET statement assigns the string `123 maple avenue` to macro variable ADDRESS. The second %LET statement uses the %SCAN function to search the source (first argument) and retrieve the first word (second argument). Because the macro processor executes the %SCAN function before it stores the value, the value of FRSTWORD is the string `123`.

For more information about %SCAN, see "%SCAN Macro Function" on page 337. For more information about macro functions, see Chapter 12, "Macro Language Elements," on page 189.

4

Macro Processing

Macro Processing

This section describes macro processing and shows the typical pattern that SAS follows to process a program containing macro elements. For most macro programming, you do not need this level of detail. It is provided to help you understand what is going on behind the scenes.

Defining and Calling Macros

Macros are compiled programs that you can call in a submitted SAS program or from a SAS command prompt. Like macro variables, you generally use macros to generate text. However, macros provide additional capabilities:

- Macros can contain programming statements that enable you to control how and when text is generated.
- Macros can accept parameters. You can write generic macros that can serve a number of uses.

To compile a macro, you must submit a macro definition. The following is the general form of a macro definition:

%MACRO *macro_name*;

<*macro_text*>

%MEND <*macro_name*>;

macro_name is a unique SAS name that identifies the macro and *macro_text* is any combination of macro statements, macro calls, text expressions, or constant text.

When you submit a macro definition, the macro processor compiles the definition and produces a member in the session catalog. The member consists of compiled macro program statements and text. The distinction between compiled items and noncompiled (text) items is important for macro execution. Examples of text items include:

- macro variable references
- nested macro calls
- macro functions, except %STR and %NRSTR
- arithmetic and logical macro expressions
- text to be written by %PUT statements

When you want to call the macro, you use the form

%*macro_name*.

You can put a DATA step in a macro:

```
%macro ds;
   data _null_;
      put "Fred";
   run;
%mend;

%ds;
```

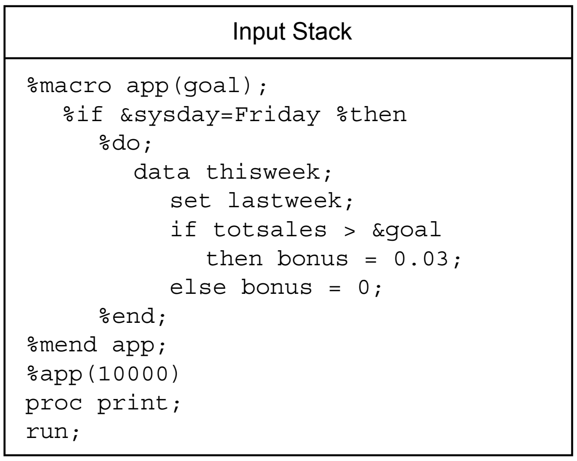
Note: If your macro_text contains passwords that you want to prevent from being revealed in the SAS log, redirect the SAS log to a file. For more information, see "PRINTTO Procedure" in *Base SAS Procedures Guide*.

For more information, see "%MACRO Macro Statement".

How the Macro Processor Compiles a Macro Definition

When you submit a SAS program, the contents of the program goes to an area of memory called the input stack. The example program in the following figure contains a macro definition, a macro call, and a PROC PRINT step. This section illustrates how the macro definition in the example program is compiled and stored.

Figure 4.1 *The Macro APP*


Input Stack

```
%macro app(goal);
   %if &sysday=Friday %then
      %do;
         data thisweek;
            set lastweek;
            if totsales > &goal
               then bonus = 0.03;
            else bonus = 0;
      %end;
%mend app;
%app(10000)
proc print;
run;
```

Using the same process described in Chapter 2, "SAS Programs and Macro Processing," on page 17 the word scanner begins tokenizing the program. When the word scanner detects % followed by a nonblank character in the first token, it triggers the macro processor. The macro processor examines the token and recognizes the beginning of a macro definition. The macro processor pulls tokens from the input stack and compiles until the %MEND statement terminates the macro definition (Figure 4.2 on page 48).

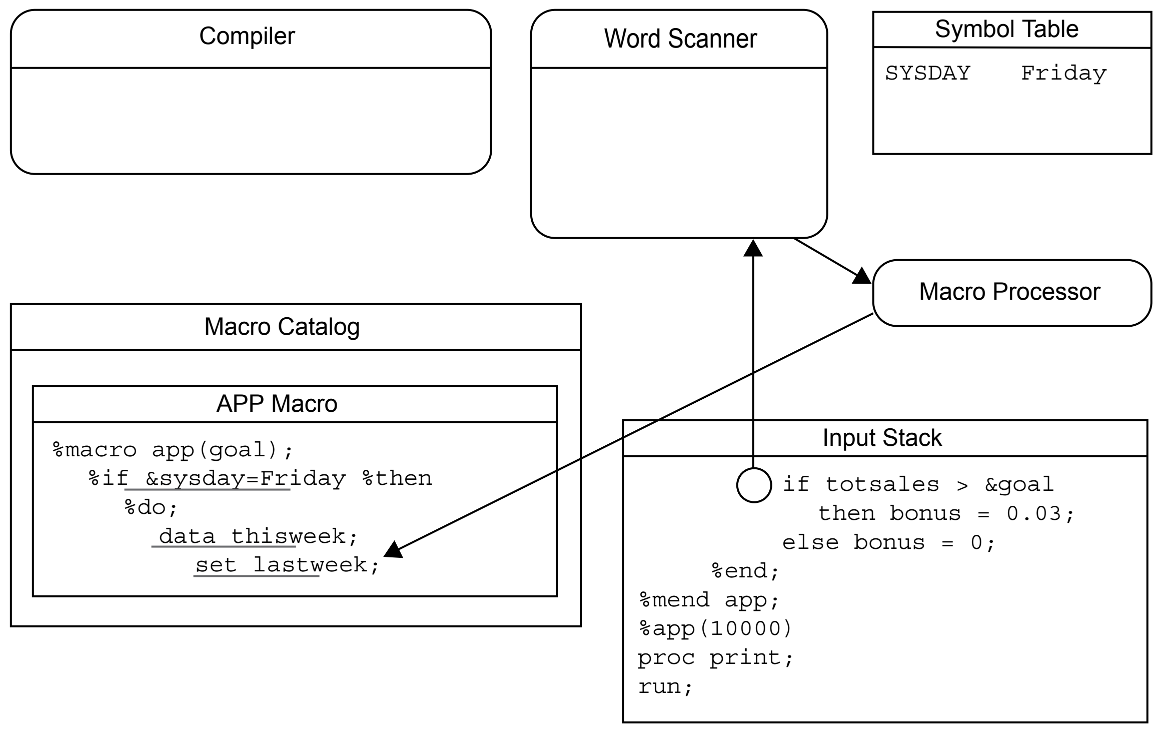
During macro compilation, the macro processor does the following:

- creates an entry in the session catalog
- compiles and stores all macro program statements for that macro as macro instructions
- stores all noncompiled items in the macro as text

Note: Text items are underlined in the illustrations in this section.

If the macro processor detects a syntax error while compiling the macro, it checks the syntax in the rest of the macro and issues messages for any additional errors that it finds. However, the macro processor does not store the macro for execution. A macro that the macro processor compiles but does not store is called a *dummy macro*.

Figure 4.2 Macro APP in the Input Stack

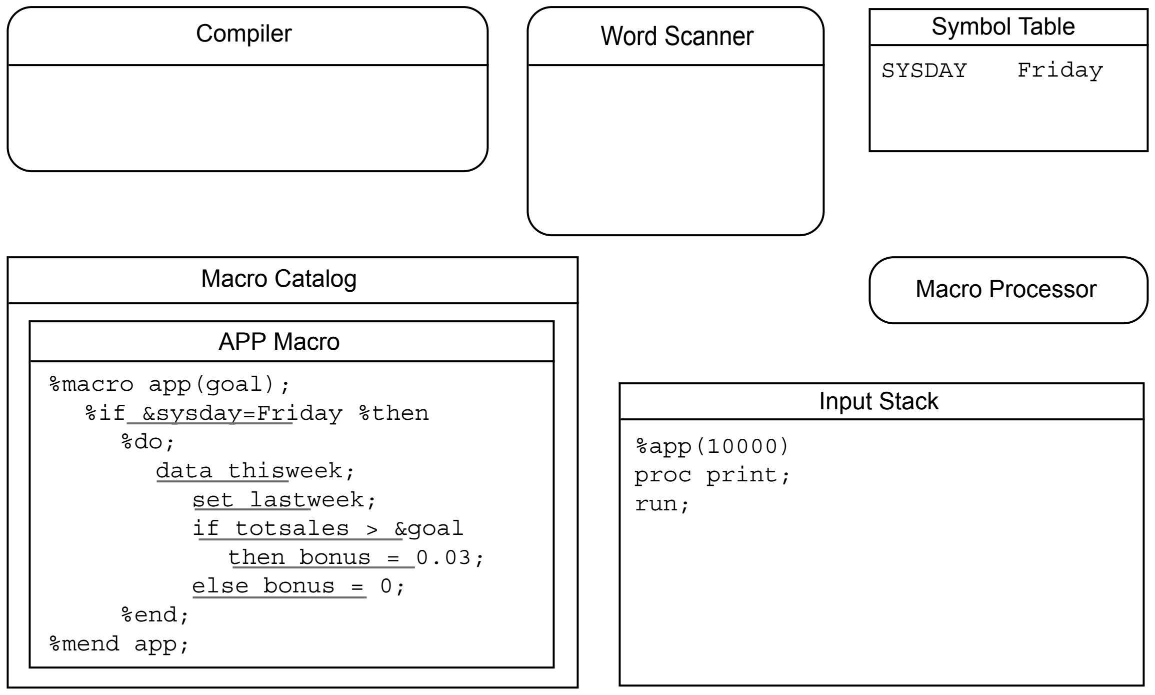


```
%macro app(goal);
   %if &sysday=Friday %then
      %do;
         data thisweek;
            set lastweek;
```

```
           if totsales > &goal
              then bonus = 0.03;
           else bonus = 0;
      %end;
%mend app;
%app(10000)
proc print;
run;
```

In this example, the macro definition is compiled and stored successfully. (See the following figure.) For the sake of illustration, the compiled APP macro looks like the original macro definition that was in the input stack. The entry would actually contain compiled macro instructions with constant text. The constant text in this example is underlined.

Figure 4.3 *The Compiled Macro APP*



Symbol Table:

```
SYSDAY    Friday
```

APP Macro:

```
%macro app(goal);
   %if &sysday=Friday %then
      %do;
         data thisweek;
            set lastweek;
            if totsales > &goal
               then bonus = 0.03;
            else bonus = 0;
      %end;
%mend app;
```

Input Stack:

```
%app(10000)
proc print;
run;
```

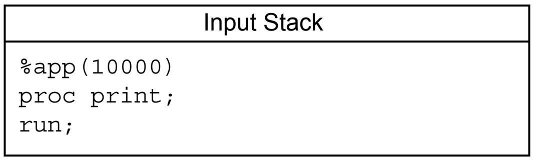
How the Macro Processor Executes a Compiled Macro

Macro execution begins with the macro processor opening the SASMacr catalog to read the appropriate macro entry. As the macro processor executes the compiled instructions in the macro entry, it performs a series of simple repetitive actions. During macro execution, the macro processor does the following:

- executes compiled macro program instructions
- places noncompiled constant text on the input stack
- waits for the word scanner to process the generated text
- resumes executing compiled macro program instructions

To continue the example from the previous section, the following figure shows the lines remaining in the input stack after the macro processor compiles the macro definition APP.

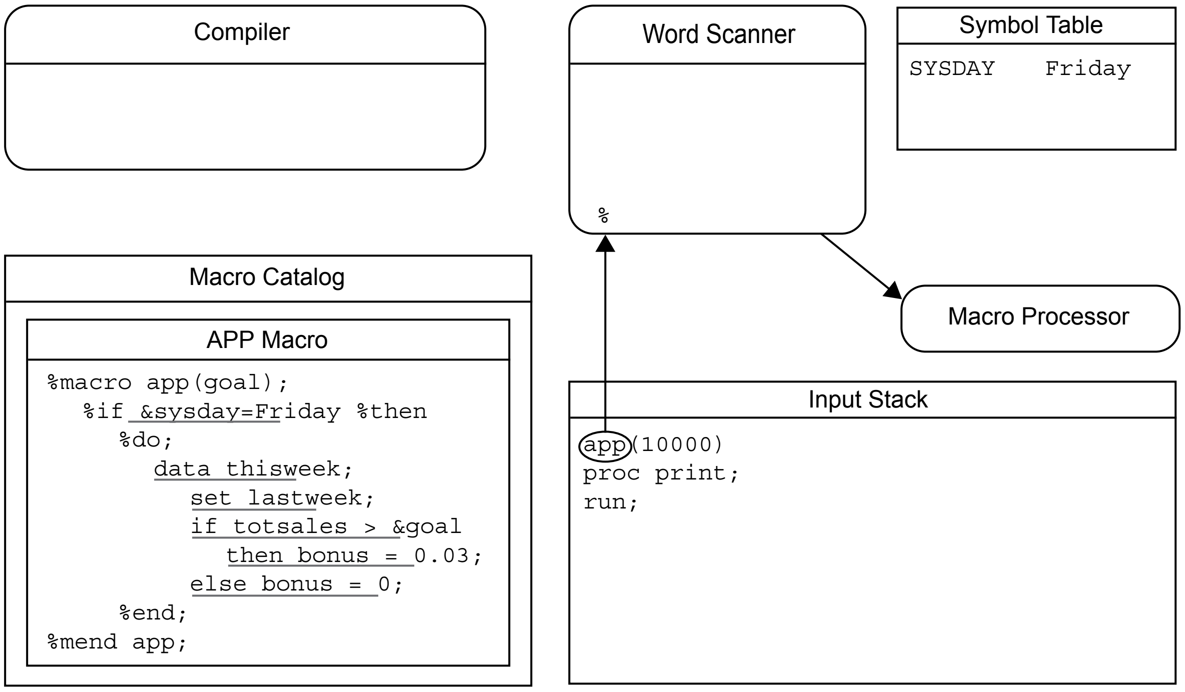
Figure 4.4 The Macro Call in the Input Stack



```
%app(10000)
proc print;
run;
```

The word scanner examines the input stack and detects % followed by a nonblank character in the first token. It triggers the macro processor to examine the token.

Figure 4.5 Macro Call Entering Word Queue



```
%macro app(goal);
   %if &sysday=Friday %then
      %do;
         data thisweek;
            set lastweek;
            if totsales > &goal
               then bonus = 0.03;
            else bonus = 0;
      %end;
%mend app;
```

```
app(10000)
proc print;
run;
```

The macro processor recognizes a macro call and begins to execute macro APP, as follows:

1 The macro processor creates a local symbol table for the macro. The macro processor examines the previously compiled definition of the macro. If there are any parameters, variable declarations, or computed GOTO statements in the macro definition, the macro processor adds entries for the parameters and variables to the newly created local symbol table.

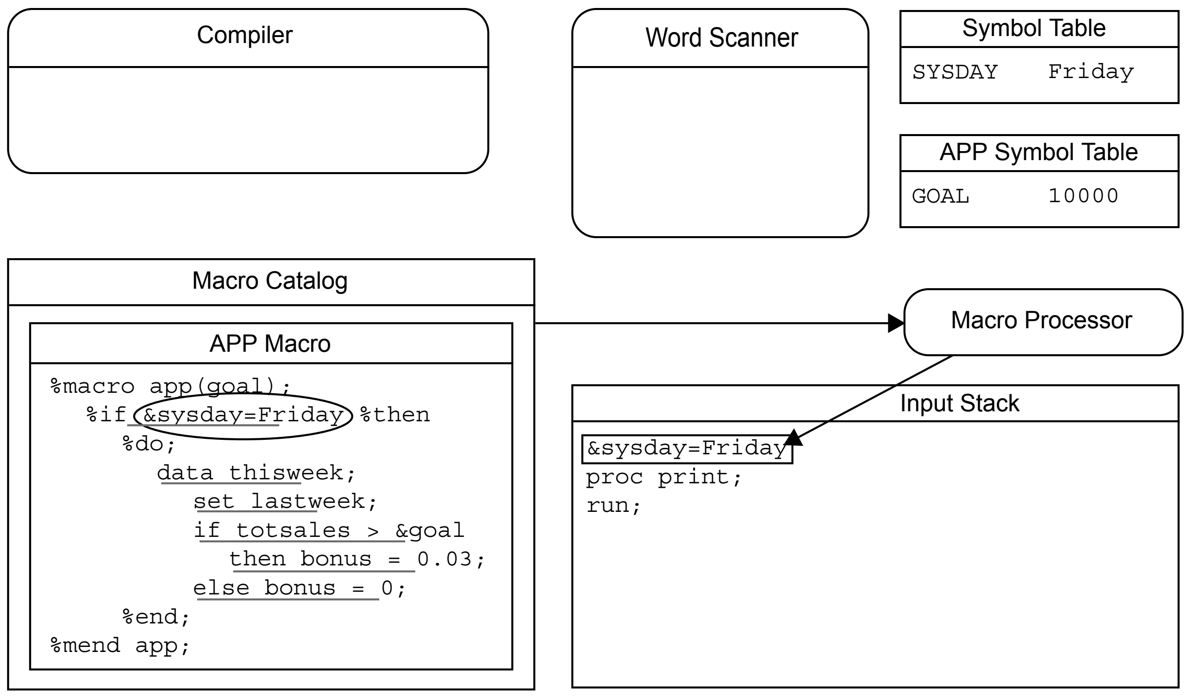
2 The macro processor further examines the previously compiled macro definition for parameters to the macro. If no parameters were defined in the macro definition, the macro processor begins to execute the compiled instructions of the macro. If any parameters were contained in the definition, the macro processor removes tokens from the input stack to obtain values for positional parameters and non-default values for keyword parameters. The values for parameters found in the input stack are placed in the appropriate entry in the local symbol table.

Note: Before executing any compiled instructions, the macro processor removes only enough tokens from the input stack to ensure that any tokens that are supplied by the user and pertain to the macro call have been removed.

3 The macro processor encounters the compiled %IF instruction and recognizes that the next item will be text containing a condition.

4 The macro processor places the text `&sysday=Friday` on the input stack ahead of the remaining text in the program. (See the following figure). The macro processor waits for the word scanner to tokenize the generated text.

Figure 4.6 *Text for %IF Condition on Input Stack*



```
%macro app(goal);
   %if &sysday=Friday %then
      %do;
         data thisweek;
            set lastweek;
            if totsales > &goal
               then bonus = 0.03;
            else bonus = 0;
      %end;
%mend app;
```

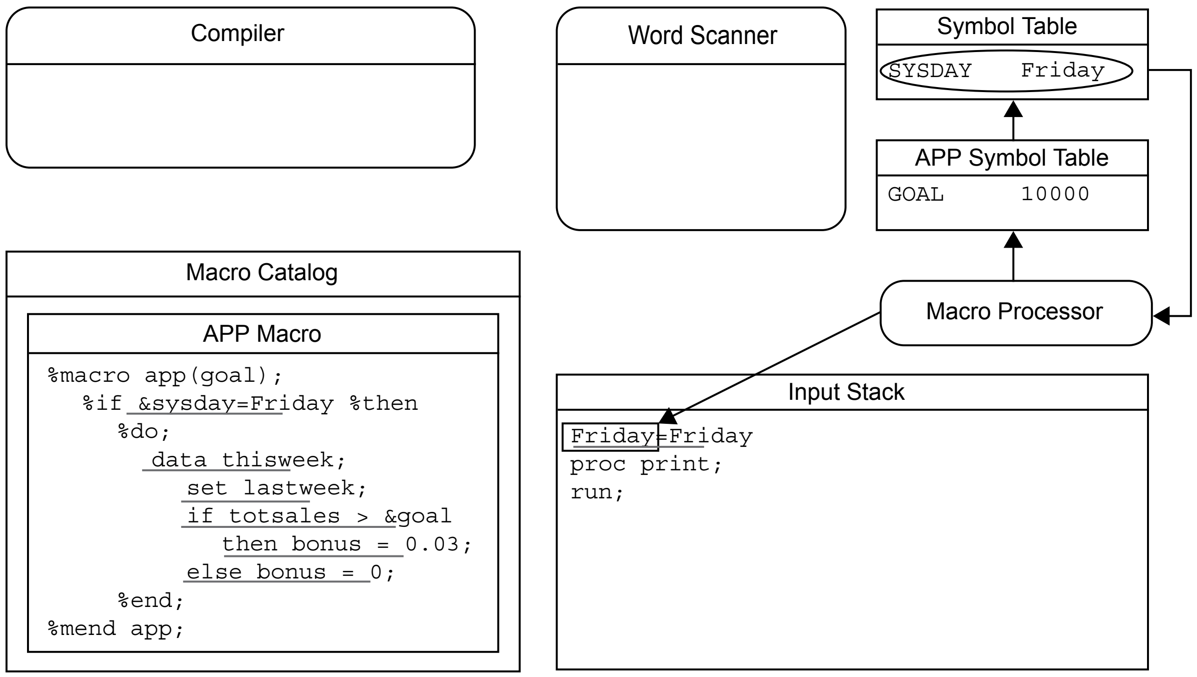
```
&sysday=Friday
proc print;
run;
```

1 The word scanner starts tokenizing the generated text, recognizes an ampersand followed by nonblank character in the first token, and triggers the macro processor.

2 The macro processor examines the token and finds a possible macro variable reference, &SYSDAY. The macro processor first searches the local APP symbol table for a matching entry and then the global symbol table. When the macro processor finds the entry in the global symbol table, it replaces macro variable in the input stack with the value `Friday`. (See the following figure.)

3 The macro processor stops and waits for the word scanner to tokenize the generated text.

Figure 4.7 Input Stack After Macro Variable Reference Is Resolved

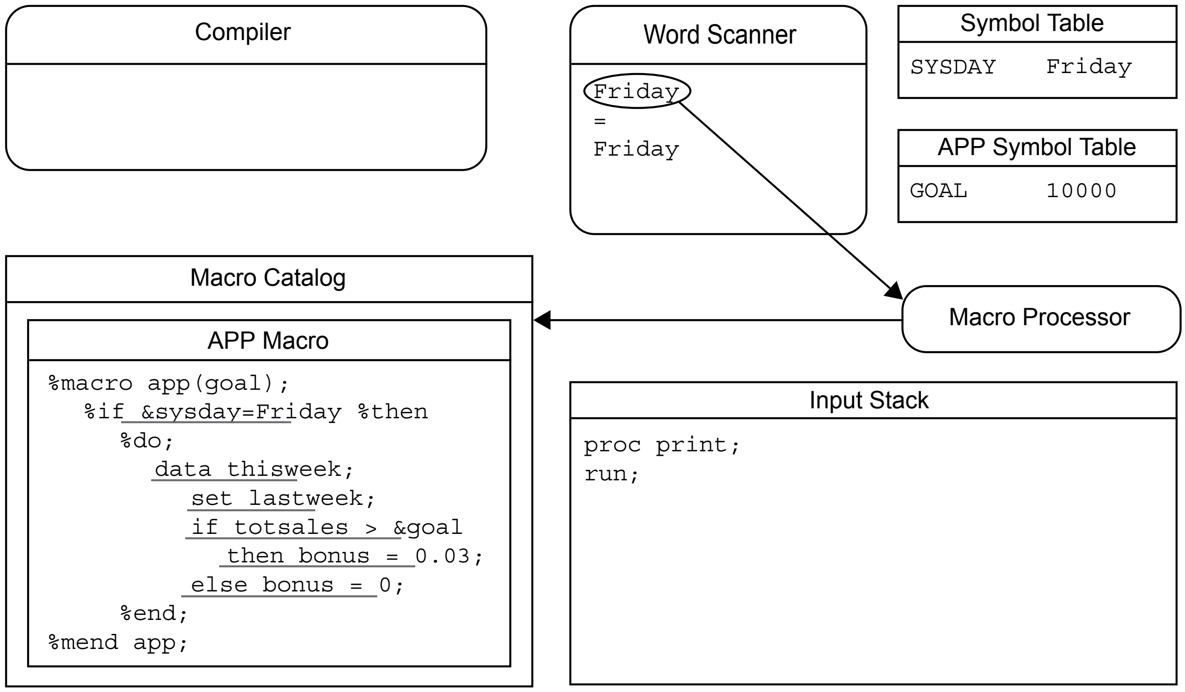


```
%macro app(goal);
   %if &sysday=Friday %then
      %do;
         data thisweek;
            set lastweek;
            if totsales > &goal
               then bonus = 0.03;
            else bonus = 0;
      %end;
%mend app;
```

1 The word scanner then read `Friday=Friday` from the input stack.

2 The macro processor evaluates the expression `Friday=Friday` and, because the expression is true, proceeds to the %THEN and %DO instructions.

Figure 4.8 Macro Processor Receives the Condition



```
%macro app(goal);
   %if &sysday=Friday %then
      %do;
         data thisweek;
            set lastweek;
            if totsales > &goal
               then bonus = 0.03;
            else bonus = 0;
      %end;
%mend app;
```

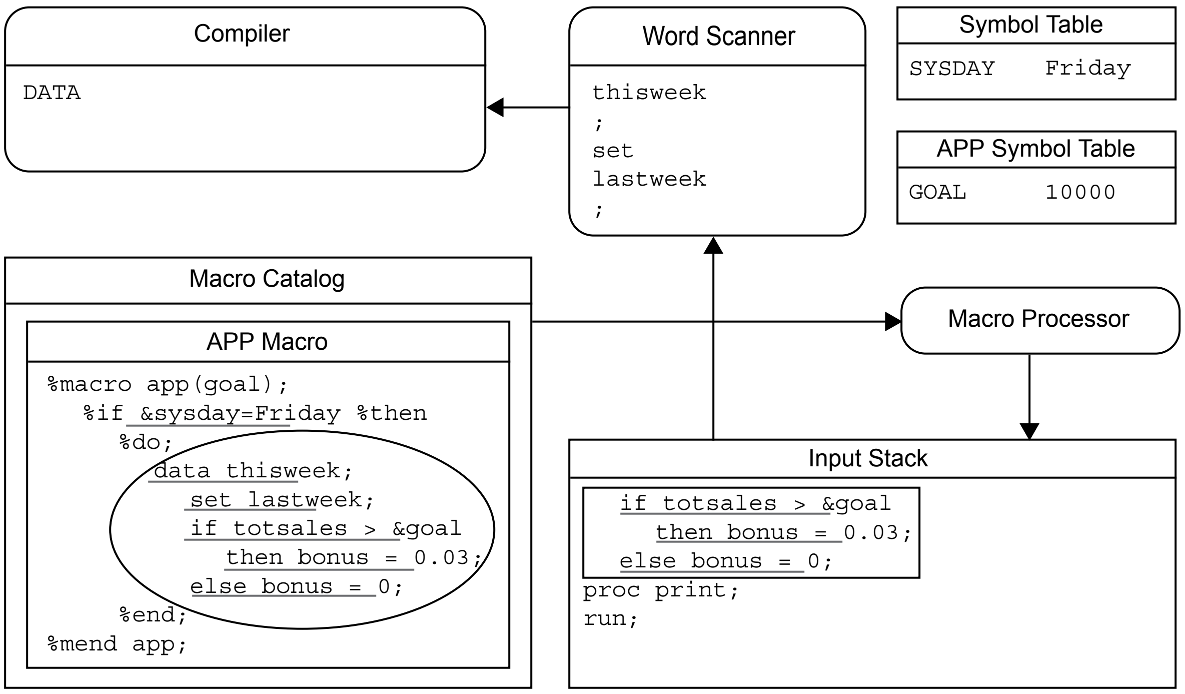
1 The macro processor executes the compiled %DO instructions and recognizes that the next item is text.

2 The macro processor places the text on top of the input stack and waits for the word scanner to begin tokenization.

3 The word scanner reads the generated text from the input stack, and tokenizes it.

4 The word scanner recognizes the beginning of a DATA step, and triggers the compiler to begin accepting tokens. The word scanner transfers tokens to the compiler from the top of the stack.

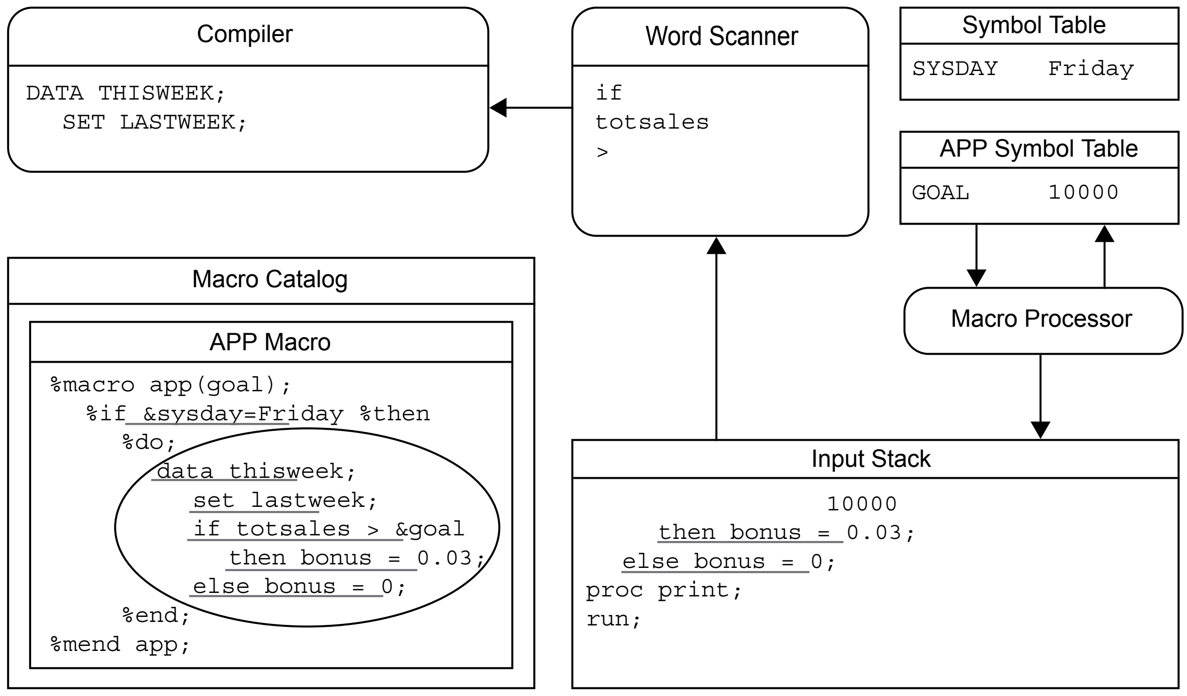
Figure 4.9 *Generated Text on Top of Input Stack*



1 When the word scanner detects & followed by a nonblank character (the macro variable reference &GOAL), it triggers the macro processor.

2 The macro processor looks in the local APP symbol table and resolves the macro variable reference &GOAL to `10000`. The macro processor places the value on top of the input stack, ahead of the remaining text in the program.

Figure 4.10 The Word Scanner Reads Generated Text



1 The word scanner resumes tokenization. When it has completed tokenizing the generated text, it triggers the macro processor.

2 The macro processor resumes processing the compiled macro instructions. It recognizes the end of the %DO group at the %END instruction and proceeds to %MEND.

3 the macro processor executes the %MEND instruction, removes the local symbol table APP, and macro APP ceases execution.

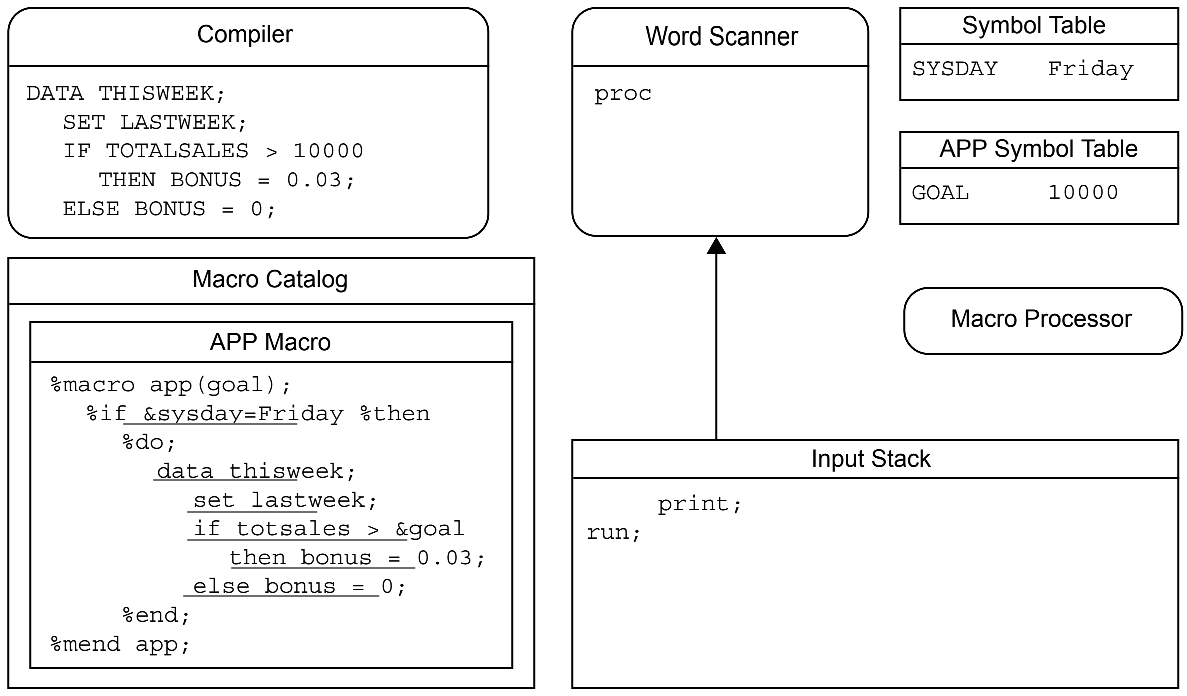
4 The macro processor triggers the word scanner to resume tokenization.

5 The word scanner reads the first token in the input stack (PROC), recognizes the beginning of a step boundary, and triggers the DATA step compiler.

6 The compiled DATA step is executed, and the DATA step compiler is cleared.

7 The word scanner signals the PRINT procedure (a separate executable not illustrated), which pulls the remaining tokens.

Figure 4.11 The Remaining Statements Are Compiled and Executed



Summary of Macro Processing

The previous sections illustrate the relationship between macro compilation and execution and DATA step compilation and execution. The relationship contains a pattern of simple repetitive actions. These actions begin when text is submitted to the input stack and the word scanner begins tokenization. At times the word scanner waits for the macro processor to perform an activity, such as searching the symbol tables or compiling a macro definition. If the macro processor generates text during its activity, then it pauses while the word scanner tokenizes the text and sends the tokens to the appropriate target. These tokens might trigger other actions in parts of SAS, such as the DATA step compiler, the command processor, or a SAS procedure. If any of these actions occur, the macro processor waits for these actions to be completed before resuming its activity. When the macro processor stops, the word scanner resumes tokenization. This process continues until the entire program has been processed.

5

Scopes of Macro Variables

Scopes of Macro Variables

Every macro variable has a *scope*. A macro variable's scope determines how it is assigned values and how the macro processor resolves references to it.

Two types of scopes exist for macro variables: *global* and *local*. Global macro variables exist for the duration of the Compute Server session and can be referenced anywhere (except CARDS and DATALINES) in the program—either inside or outside a macro. Local macro variables exist only during the execution of the macro in which the variables are created and have no meaning outside the defining macro.

Scopes can be nested, like boxes within boxes. For example, suppose you have a macro A that creates the macro variable LOC1 and a macro B that creates the macro variable LOC2. If the macro B is nested (executed) within the macro A, LOC1 is local to both A and B. However, LOC2 is local only to B.

Macro variables are stored in *symbol tables*, which list the macro variable name and its value. There is a global symbol table, which stores all global macro variables. Local macro variables are stored in a local symbol table that is created at the beginning of the execution of a macro.

You can use the %SYMEXIST function to indicate whether a macro variable exists. For more information, see "%SYMEXIST Macro Function" on page 349.

Global Macro Variables

The following code illustrates the global symbol table during execution of the following program:

```
%let county=Clark;

%macro concat;
   data _null_;
      length longname $20;
      longname="&county"||" County";
      put longname;
   run;
%mend concat;

%concat
```

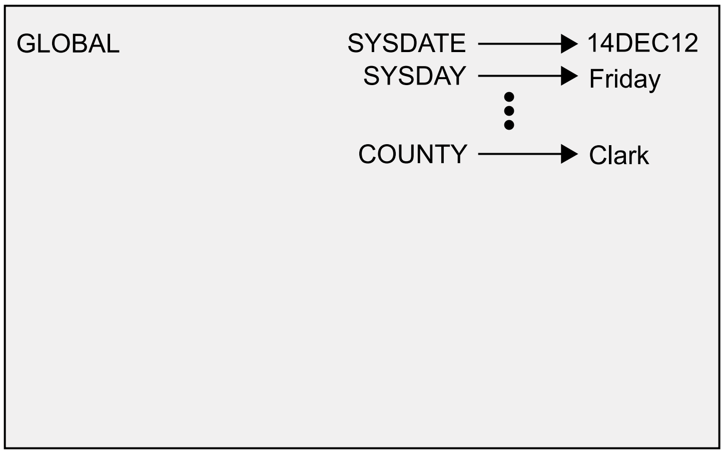
Calling the macro CONCAT produces the following statements:

```
data _null_;
      length longname $20;
      longname="Clark"||" County";
      put longname;
run;
```

The PUT statement writes the following to the SAS log:

```
Clark County
```

Figure 5.1 Global Symbol Table



Global macro variables include the following:

- all automatic macro variables except SYSPBUFF. For more information about SYSPBUFF and other automatic macro variables, see "Automatic Macro Variables" on page 200.
- macro variables created outside of any macro.
- macro variables created in %GLOBAL statements. For more information about the %GLOBAL statement, see "Creating Global Macro Variables" on page 74.
- most macro variables created by the CALL SYMPUT routine. For more information about the CALL SYMPUT routine, see "Special Cases of Scope with the CALL SYMPUT Routine" on page 77.

You can create global macro variables anytime during a Compute Server session or job. Except for some automatic macro variables, you can change the values of global macro variables anytime during a Compute Server session or job.

In most cases, once you define a global macro variable, its value is available to you anywhere in the session or job and can be changed anywhere. So, a macro variable referenced inside a macro definition is global if a global macro variable already exists by the same name. This action assumes that the variable is not specifically defined as local with the %LOCAL statement or in a parameter list. The new macro variable definition simply updates the existing global one. The following are exceptions that prevent you from referencing the value of a global macro variable:

- when a macro variable exists both in the global symbol table and in the local symbol table, you cannot reference the global value from within the macro that contains the local macro variable. In this case, the macro processor finds the local value first and uses it instead of the global value.
- if you create a macro variable in the DATA step with the SYMPUT routine, you cannot reference the value with an ampersand until the program reaches a step boundary. For more information about macro processing and step boundaries, see "%SYMGLOBL Macro Function" on page 350.

You can create a read-only global macro variable and assign a specified value to it using the READONLY option in a %GLOBAL statement. Existing macro variables cannot be made read-only. The value of the variable cannot be changed, and the variable cannot be deleted. All read-only macro variables persist until the scope in which they exist is deleted. For more information, see “%GLOBAL Macro Statement” on page 393.

You can use the %SYMGLOBL function to indicate whether an existing macro variable resides in the global symbol table. For more information, see “%SYMGLOBL Macro Function” on page 350.

Local Macro Variables

Local macro variables are defined within an individual macro. Each macro that you invoke creates its own local symbol table. Local macro variables exist only as long as a particular macro executes. When the macro stops executing, all local macro variables for that macro cease to exist.

The following code illustrates the local symbol table during the execution of the following program.

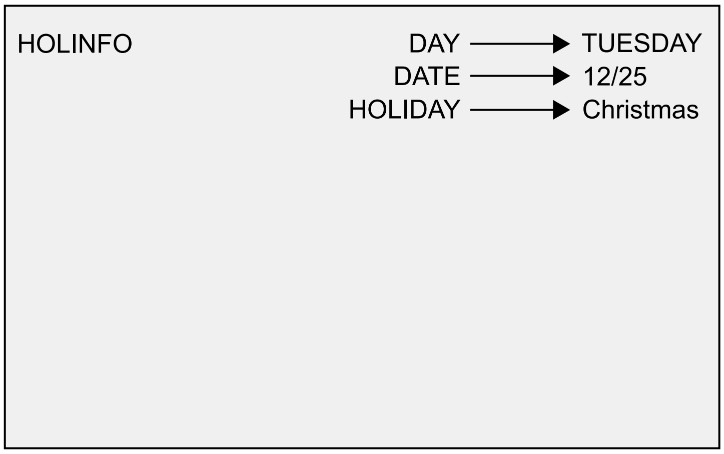
```
%macro holinfo(day,date);
   %let holiday=Christmas;
   %put *** Inside macro: ***;
   %put *** &holiday occurs on &day, &date, 2012. ***;
%mend holinfo;

%holinfo(Tuesday,12/25)

%put *** Outside macro: ***;
%put *** &holiday occurs on &day, &date, 2012. ***;
```

The %PUT statements write the following to the SAS log:

```
*** Inside macro: ***
*** Christmas occurs on Tuesday, 12/25, 2012. ***
66
67   %put *** Outside macro: ***;
*** Outside macro: ***
68   %put *** &holiday occurs on &day, &date, 2012. ***;
WARNING: Apparent symbolic reference HOLIDAY not resolved.
WARNING: Apparent symbolic reference DAY not resolved.
WARNING: Apparent symbolic reference DATE not resolved.
*** &holiday occurs on &day, &date, 2012. ***
```

As you can see from the log, the local macro variables DAY, DATE, and HOLIDAY resolve inside the macro. But outside the macro, they do not exist and therefore do not resolve.

Figure 5.2 *Local Symbol Table*



A macro's local symbol table is empty until the macro creates at least one macro variable. A local symbol table can be created by any of the following:

- the presence of one or more macro parameters
- a %LOCAL statement
- macro statements that define macro variables, such as %LET and the iterative %DO statement (if the variable does not already exist globally or a %GLOBAL statement is not used)

Note: Macro parameters are always local to the macro that defines them. You cannot make macro parameters global. (Although, you can assign the value of the parameter to a global variable. For more information, see "Creating Global Variables Based on the Value of Local Variables" on page 76.)

When you invoke one macro inside another, you create nested scopes. Because you can have any number of levels of nested macros, your programs can contain any number of levels of nested scopes.

You can create a read-only local macro variable and assign a specified value to it using the READONLY option in a %LOCAL statement. Existing macro variables cannot be made read-only. The value of the variable cannot be changed, and the variable cannot be deleted. All read-only macro variables persist until the scope in which they exist is deleted. For more information, see "%LOCAL Macro Statement" on page 403.

You can use the %SYMLOCAL function to indicate whether an existing macro variable resides in an enclosing local symbol table. For more information, see the "%SYMLOCAL Macro Function" on page 351.

Writing the Contents of Symbol Tables to the SAS Log

While developing your macros, you might find it useful to write all or part of the contents of the global and local symbol tables to the SAS log. To do so, use the %PUT statement with one of the following options:

ALL
: describes all currently defined macro variables, regardless of scope. This output includes user-defined global and local variables as well as automatic macro variables. Scopes are listed in the order of innermost to outermost.

AUTOMATIC
: describes all automatic macro variables. The scope is listed as AUTOMATIC. All automatic macro variables are global except SYSPBUFF. For more information about specific automatic macro variables, see "Automatic Macro Variables" on page 200.

GLOBAL
: describes all global macro variables that were not created by the macro processor. The scope is listed as GLOBAL. Automatic macro variables are not listed.

LOCAL
: describes user-defined local macro variables defined within the currently executing macro. The scope is listed as the name of the macro in which the macro variable is defined.

READONLY
: describes all user-defined read-only macro variables, regardless of scope. The scope is either GLOBAL, for global macro variables, or the name of the macro in which the macro variable is defined.

USER
: describes all user-defined macro variables, regardless of scope. The scope is either GLOBAL, for global macro variables, or the name of the macro in which the macro variable is defined.

WRITABLE
: describes all user-defined read and write macro variables, regardless of scope. The scope is either GLOBAL, for global macro variables, or the name of the macro in which the macro variable is defined.

For example, consider the following program:

```
%let origin=Asia;

%macro cars(type=);
   data _null_;
      set sashelp.cars;
      where type="&type" and origin="&origin";
   run;

   %put _user_;
%mend cars;

%cars(type=Sedan)
```

The %PUT statement preceding the %MEND statement writes to the SAS log the scopes, names, and values of all user-generated macro variables:

```
CARS TYPE Sedan
GLOBAL ORIGIN Asia
```

Because TYPE is a macro parameter, TYPE is local to the macro CARS, with value `Sedan`. Because ORIGIN is defined in open code, it is global.

How Macro Variables Are Assigned and Resolved

Before the macro processor creates a variable, assigns a value to a variable, or resolves a variable, it searches the symbol tables to determine whether the variable already exists. The search begins with the most local scope and, if necessary, moves outward to the global scope. The request to assign or resolve a variable comes from a macro variable reference in open code (outside a macro) or within a macro.

The following figure illustrates the search order the macro processor uses when it receives a macro variable reference that requests a variable be created or assigned. The figure below illustrates the process for resolving macro variable references. Both of these figures represent the most basic type of search and do not apply in special cases, such as when a %LOCAL statement is used or the variable is created by CALL SYMPUT.

Figure 5.3 Search Order When Assigning or Creating Macro Variables

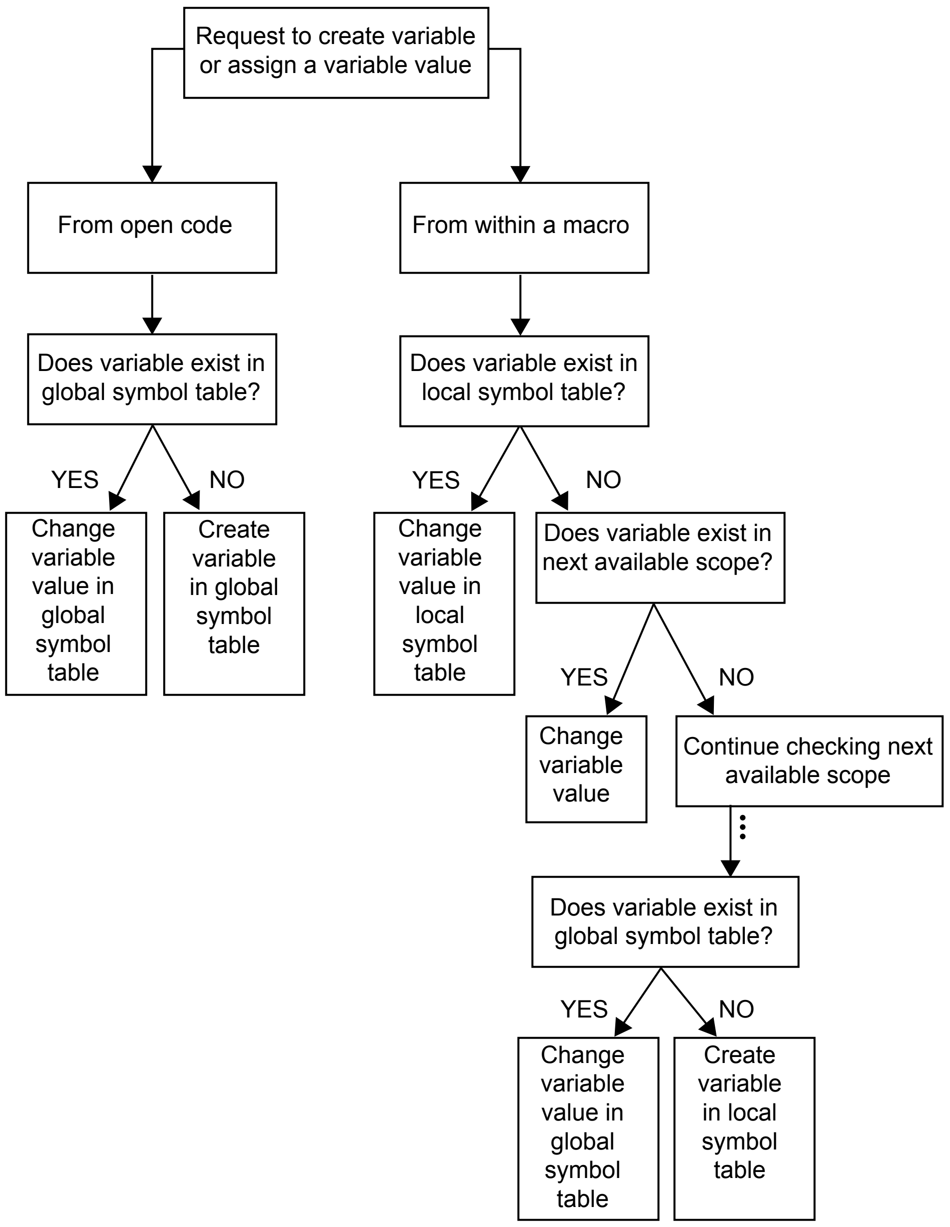
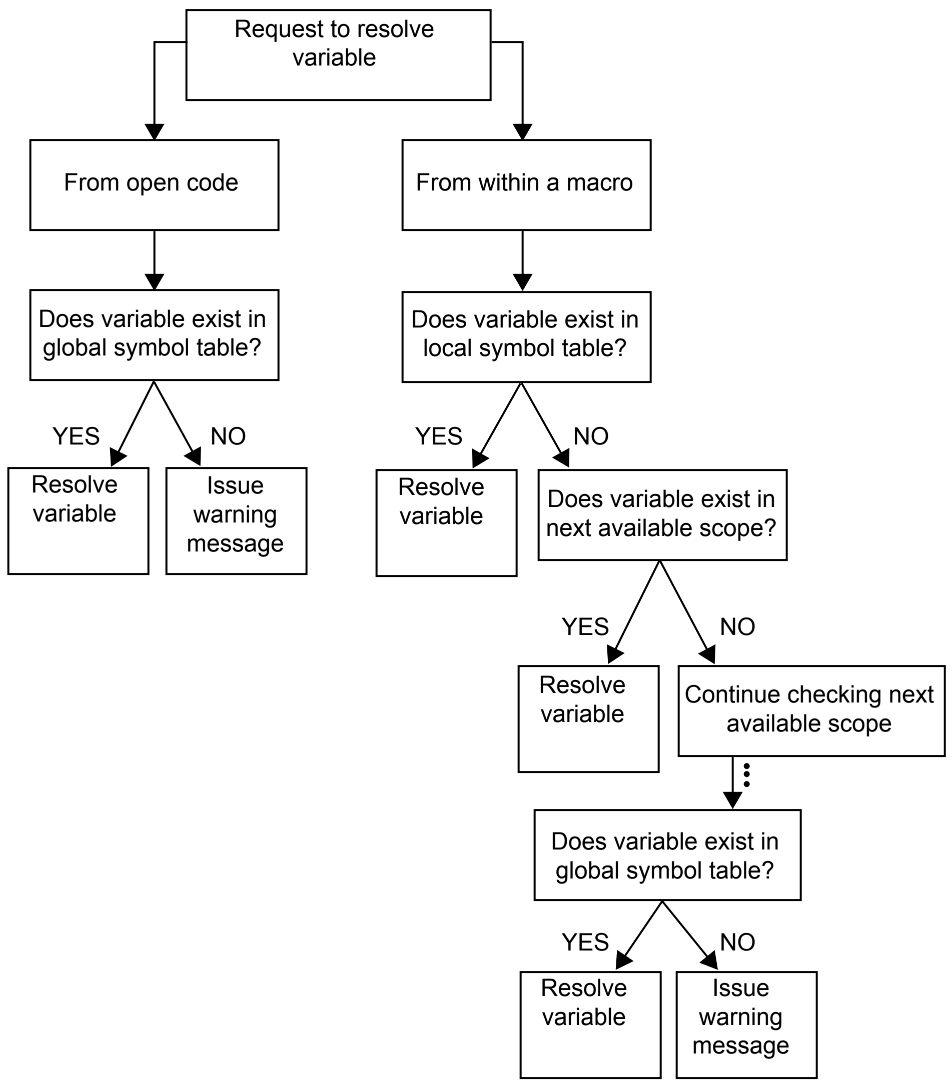

Request to create variable or assign a variable value
From open code
From within a macro
Does variable exist in global symbol table?
Does variable exist in local symbol table?
YES
NO
YES
NO
Change variable value in global symbol table
Create variable in global symbol table
Change variable value in local symbol table
Does variable exist in next available scope?
YES
NO
Change variable value
Continue checking next available scope
Does variable exist in global symbol table?
YES
NO
Change variable value in global symbol table
Create variable in local symbol table

Figure 5.4 Search Order When Resolving Macro Variable References


Request to resolve variable
From open code
From within a macro
Does variable exist in global symbol table?
YES
NO
Resolve variable
Issue warning message
Does variable exist in local symbol table?
YES
NO
Resolve variable
Does variable exist in next available scope?
YES
NO
Resolve variable
Continue checking next available scope
Does variable exist in global symbol table?
YES
NO
Resolve variable
Issue warning message

Examples of Macro Variable Scopes

Changing the Values of Existing Macro Variables

When the macro processor executes a macro program statement that can create a macro variable (such as a %LET statement), the macro processor attempts to change the value of an existing macro variable rather than create a new macro variable. The %GLOBAL and %LOCAL statements are exceptions.

To illustrate, consider the following %LET statements. Both statements assign values to the macro variable NEW:

```
%let new=inventry;
%macro name1;
   %let new=report;
   %put _local_;
%mend name1;
```

Suppose you submit the following statements:

```
%name1
%put &new;
```

Because NEW exists as a global variable, the macro processor changes the value of the variable rather than creating a new one. The macro NAME1's local symbol table remains empty.

The following figure illustrates the contents of the global and local symbol tables before, during, and after NAME1's execution.

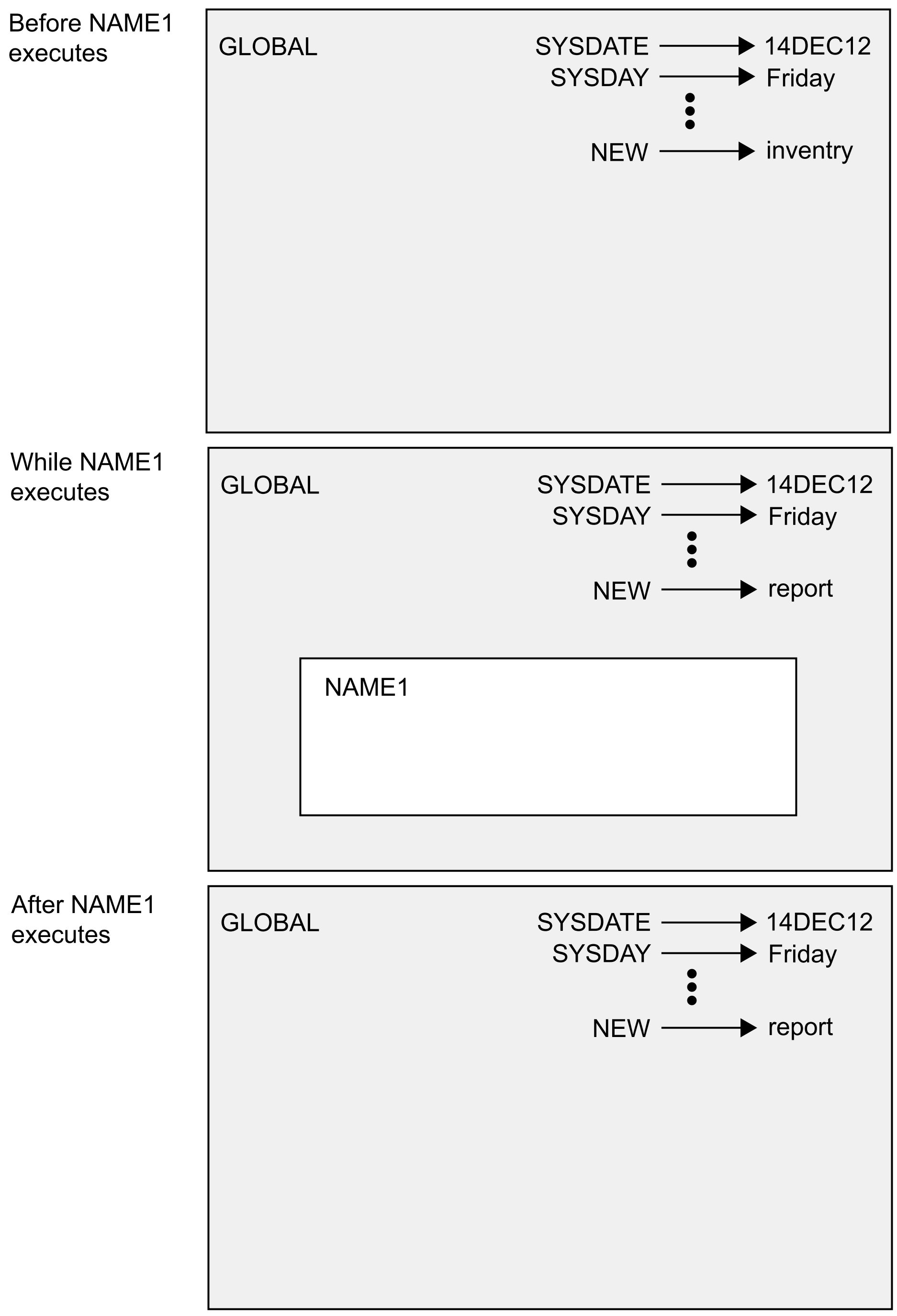


Figure 5.5 *Snapshots of Symbol Tables*

Creating Local Variables

When the macro processor executes a macro program statement that can create a macro variable, the macro processor creates the variable in the local symbol table if no macro variable with the same name is available to it. Consider the following example:

```
%let new=inventry;
%macro name2;
   %let new=report;
   %let old=warehse;
%mend name2;

%name2

data &new;
   set &old;
run;
```

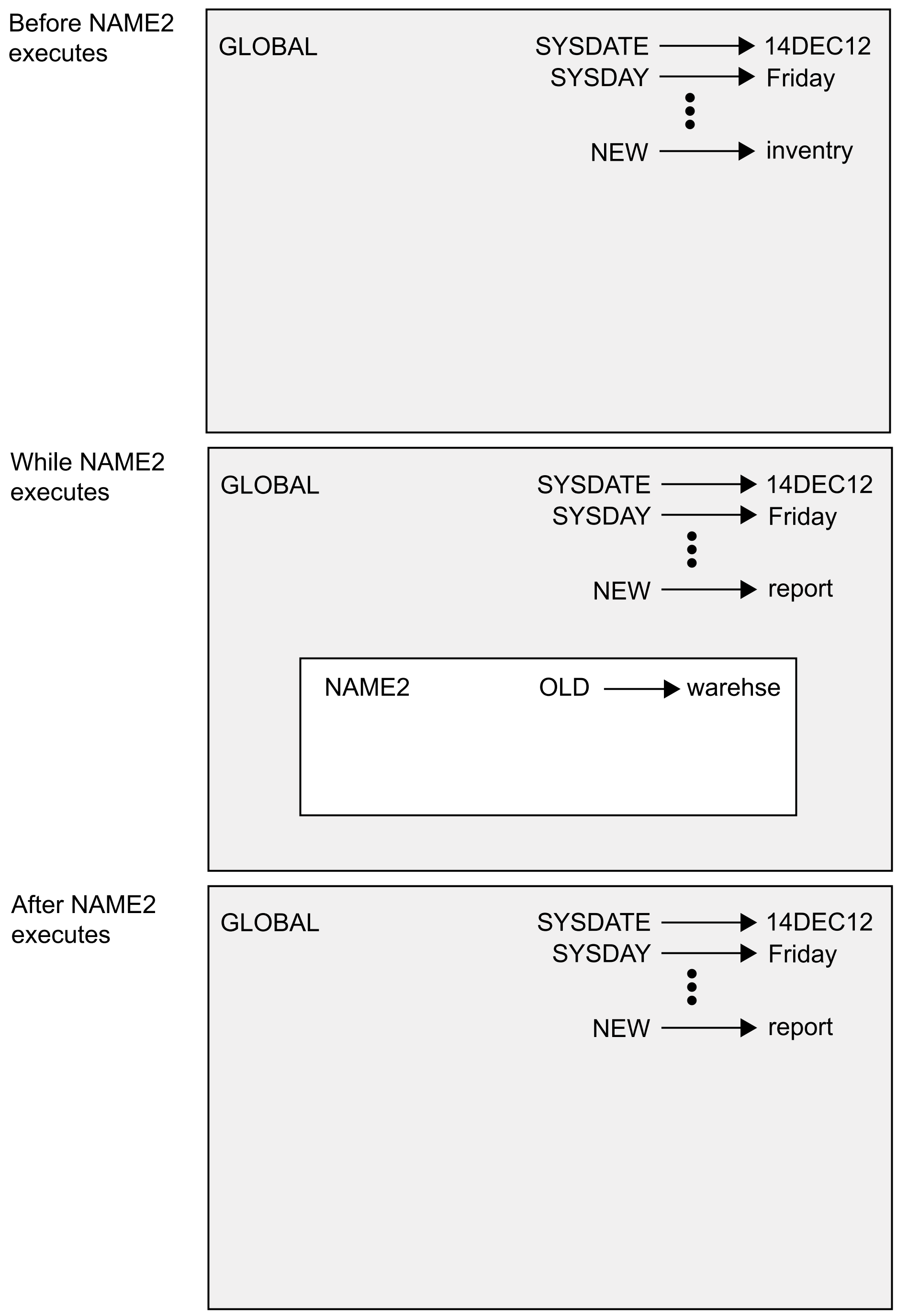
After NAME2 executes, the SAS compiler sees the following statements:

```
data report;
   set &old;
run;
```

The macro processor encounters the reference &OLD after macro NAME2 has finished executing. Thus, the macro variable OLD no longer exists. The macro processor is not able to resolve the reference and issues a warning message.

The following figure illustrates the contents of the global and local symbol tables at various stages.

Figure 5.6 *Symbol Tables at Various Stages*



But suppose you place the SAS statements inside the macro NAME2, as in the following program:

```
%let new=inventry;
%macro name2;
   %let new=report;
   %let old=warehse;
   data &new;
      set &old;
   run;
%mend name2;
```

```
%name2
```

In this case, the macro processor generates the SET statement during the execution of NAME2, and it locates OLD in NAME2's local symbol table. Therefore, executing the macro produces the following statements:

```
data report;
   set warehse;
run;
```

The same rule applies regardless of how many levels of nesting exist. Consider the following example:

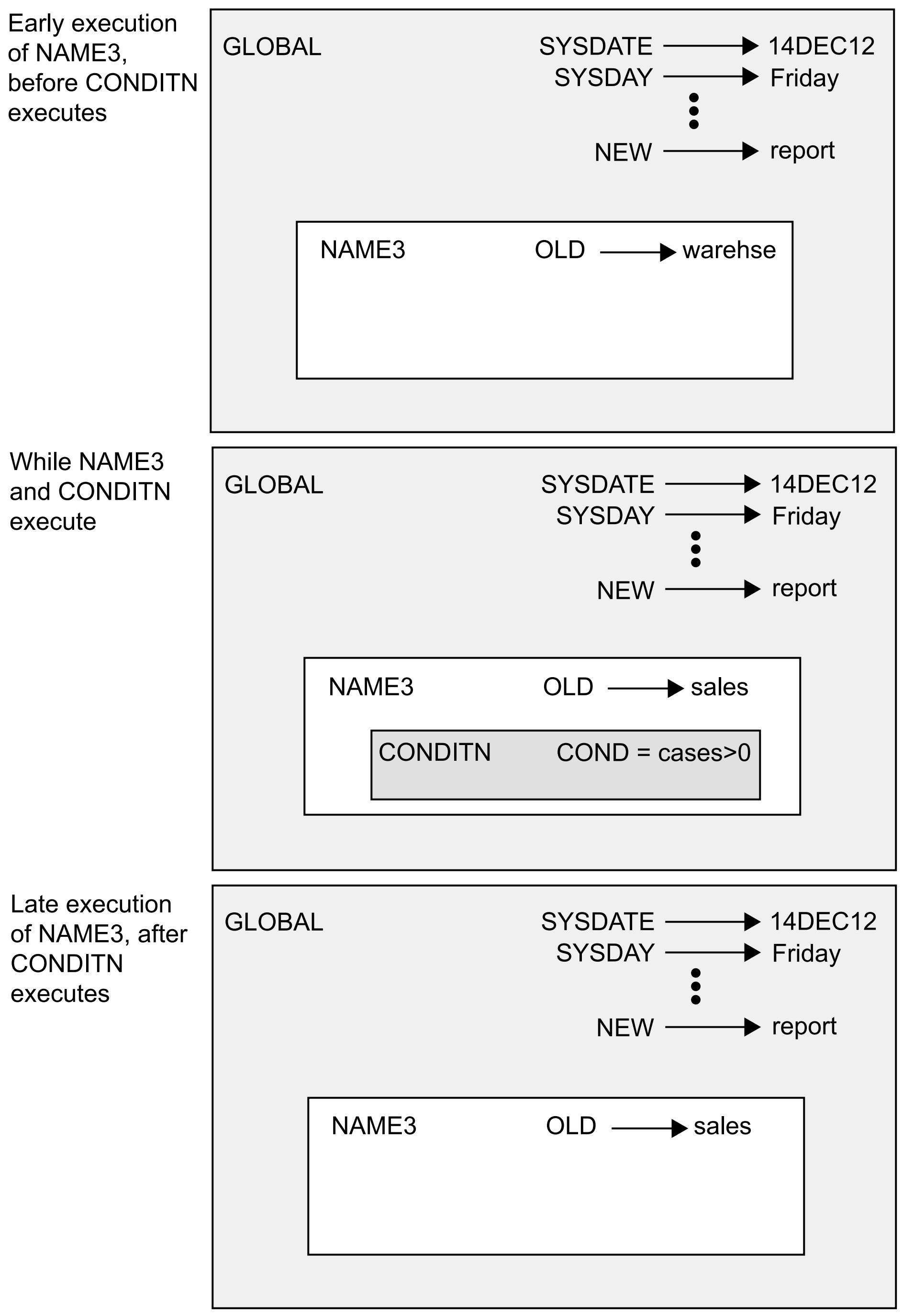
```
%let new=inventry;
%macro conditn;
   %let old=sales;
   %let cond=cases>0;
%mend conditn;

%macro name3;
   %let new=report;
   %let old=warehse;
   %conditn
      data &new;
         set &old;
         if &cond;
      run;
%mend name3;

%name3
```

The macro processor generates these statements:

```
data report;
   set sales;
   if &cond;
run;
```

CONDITN finishes executing before the macro processor reaches the reference &COND, so no variable named COND exists when the macro processor attempts to resolve the reference. Thus, the macro processor issues a warning message and generates the unresolved reference as part of the constant text and issues a warning message. The following figure shows the symbol tables at each step.

Figure 5.7 Symbol Tables Showing Two Levels of Nesting



Notice that the placement of a macro invocation is what creates a nested scope, not the placement of the macro definition. For example, invoking CONDITN from within NAME3 creates the nested scope. It is not necessary to define CONDITN within NAME3.

Forcing a Macro Variable to Be Local

At times that you need to ensure that the macro processor creates a local macro variable rather than changing the value of an existing macro variable. In this case, use the %LOCAL statement to create the macro variable.

Always make all macro variables created within macros local when you do not need their values after the macro stops executing. Debugging the large macro programs is easier if you minimize the possibility of inadvertently changing a macro variable's value. Also, local macro variables do not exist after their defining macro finishes executing, but global variables exist for the duration of the Compute Server session. Therefore, local variables use less overall storage.

Suppose you want to use the macro NAMELST to create a list of names for a VAR statement, as shown here:

```
%macro namelst(name,number);
   %do n=1 %to &number;
      &name&n
   %end;
%mend namelst;
```

You invoke NAMELST in this program:

```
%let n=North State Industries;

proc print;
   var %namelst(dept,5);
   title "Quarterly Report for &n";
run;
```

After macro execution, the SAS compiler sees the following statements:

```
proc print;
   var dept1 dept2 dept3 dept4 dept5;
   title "Quarterly Report for 6";
run;
```

The macro processor changes the value of the global variable N each time it executes the iterative %DO loop. (After the loop stops executing, the value of N is 6, as described in "%DO Macro Statement" on page 387.) To prevent conflicts, use a %LOCAL statement to create a local variable N, as shown here:

```
%macro namels2(name,number);
   %local n;
   %do n=1 %to &number;
      &name&n
   %end;
%mend namels2;
```

Now execute the same program:

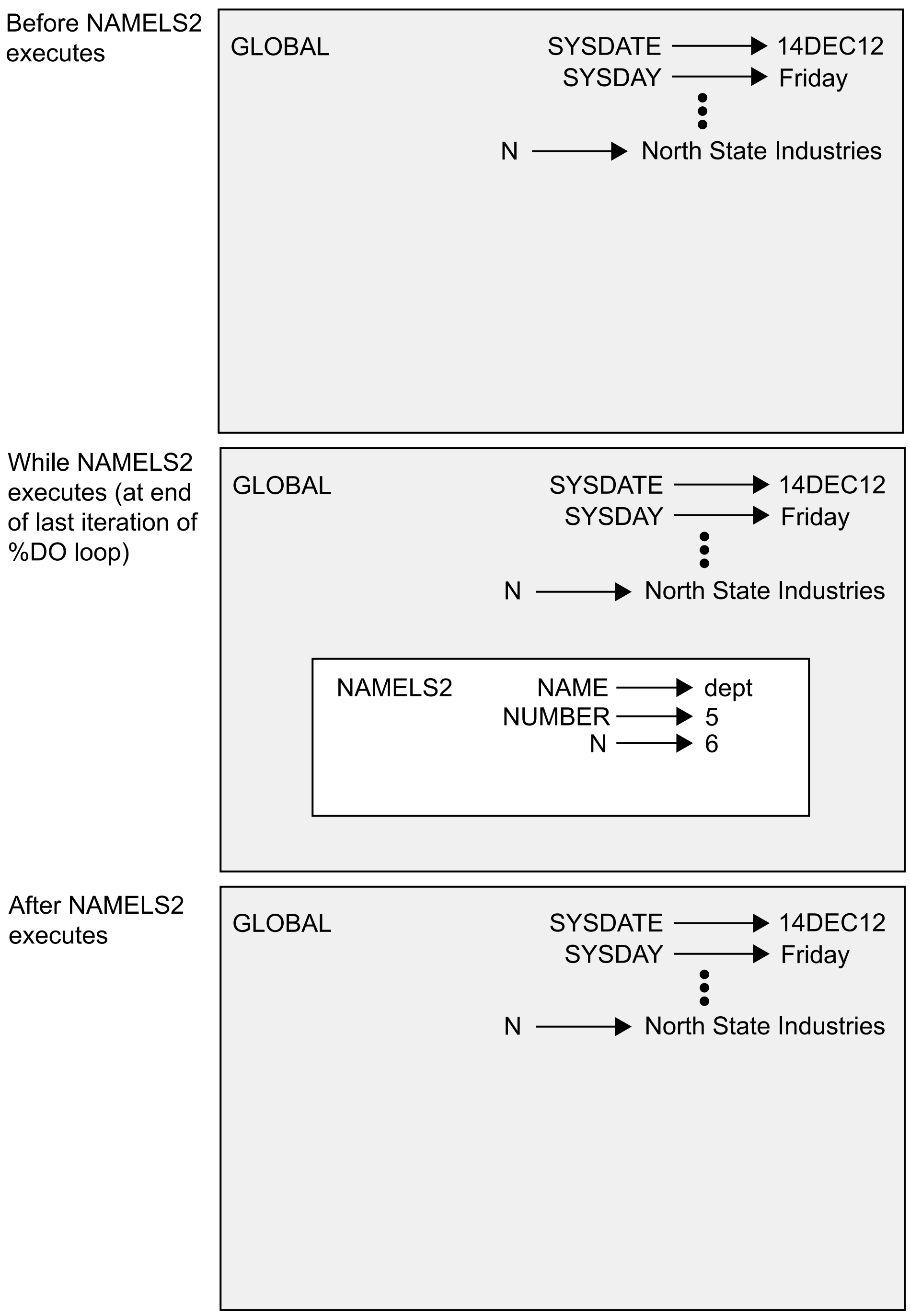
```
%let n=North State Industries;

proc print;
   var %namels2(dept,5);
   title "Quarterly Report for &n";
run;
```

The macro processor generates the following statements:

```
proc print;
   var dept1 dept2 dept3 dept4 dept5;
   title "Quarterly Report for North State Industries";
run;
```

The following figure shows the symbol tables before NAMELS2 executes, when NAMELS2 is executing, and when the macro processor encounters the reference &N in the TITLE statement.

Figure 5.8 *Symbol Tables for Global and Local Variables with the Same Name*



Creating Global Macro Variables

The %GLOBAL statement creates a global macro variable if a variable with the same name does not already exist there, regardless of what scope is current.

For example, in the following program, the macro CONDITN contains a %GLOBAL statement that creates the macro variable COND as a global variable:

```
%macro conditn;
   %global cond;
   %let old=sales;
   %let cond=cases>0;
%mend conditn;
```

Here is the rest of the program:

```
%let new=inventry;

%macro name4;
   %let new=report;
   %let old=warehse;
   %conditn
   data &new;
      set &old;
      if &cond;
   run;
%mend name4;

%name4
```

Invoking NAME4 generates these statements:

```
data report;
   set sales;
   if cases>0;
run;
```

Suppose you want to put the SAS DATA step statements outside NAME4. In this case, all the macro variables must be global for the macro processor to resolve the references. You cannot add OLD to the %GLOBAL statement in CONDITN because the %LET statement in NAME4 has already created OLD as a local variable to NAME4 by the time CONDITN begins to execute. (You cannot use the %GLOBAL statement to make an existing local variable global.)

Thus, to make OLD global, use the %GLOBAL statement before the variable reference appears anywhere else, as shown here in the macro NAME5:

```
%let new=inventry;

%macro conditn;
   %global cond;
   %let old=sales;
   %let cond=cases>0;
%mend conditn;

   %macro name5;
      %global old;
      %let new=report;
      %let old=warehse;
      %conditn
   %mend name5;

%name5
```

```
data &new;
   set &old;
   if &cond;
run;
```

Now the %LET statement in NAME5 changes the value of the existing global variable OLD rather than creating OLD as a local variable. The SAS compiler sees the following statements:

```
data report;
   set sales;
   if cases>0;
run;
```

Creating Global Variables Based on the Value of Local Variables

To use a local variable such as a parameter outside a macro, use a %LET statement to assign the value to a global variable with a different name, as in this program:

```
%macro namels3(name,number);
   %local n;
   %global g_number;
   %let g_number=&number;
   %do n=1 %to &number;
      &name&n
   %end;
%mend namels3;
```

Now invoke the macro NAMELS3 in the following the program:

```
%let n=North State Industries;

proc print;
   var %namels3(dept,5);
   title "Quarterly Report for &n";
   footnote "Survey of &g_number Departments";
run;
```

The compiler sees the following statements:

```
proc print;
   var dept1 dept2 dept3 dept4 dept5;
   title "Quarterly Report for North State Industries";
   footnote "Survey of 5 Departments";
run;
```

Special Cases of Scope with the CALL SYMPUT Routine

Overview of CALL SYMPUT Routine

Most problems with CALL SYMPUT involve the lack of a precise step boundary between the CALL SYMPUT statement that creates the macro variable and the macro variable reference that uses that variable. (For more information, see “CALL SYMPUT Routine” on page 289.) However, a few special cases exist that involve the scope of a macro variable created by CALL SYMPUT. These cases are good examples of why you should always assign a scope to a variable before assigning a value rather than relying on SAS to do it for you.

Two rules control where CALL SYMPUT creates its variables:

1. CALL SYMPUT creates the macro variable in the current symbol table available while the DATA step is executing, provided that symbol table is not empty. If it is empty (contains no local macro variables), usually CALL SYMPUT creates the variable in the closest nonempty symbol table.

2. However, there are three cases where CALL SYMPUT creates the variable in the local symbol table, even if that symbol table is empty:

   - Beginning with SAS Version 8, if CALL SYMPUT is used after a PROC SQL, the variable will be created in a local symbol table.

   - If the macro variable SYSPBUFF is created at macro invocation time, the variable will be created in the local symbol table.

   - If the executing macro contains a computed %GOTO statement, the variable will be created in the local symbol table. A computed %GOTO statement is one that uses a label that contains an & or a % in it. That is, a computed %GOTO statement contains a macro variable reference or a macro call that produces a text expression. Here is an example of a computed %GOTO statement:

     ```
     %goto &home;
     ```

The symbol table that is currently available to a DATA step is the one that exists when SAS determines that the step is complete. (SAS considers a DATA step to be complete when it encounters a RUN statement, a semicolon after data lines, or the beginning of another step.)

If an executing macro contains a computed %GOTO statement, or if the macro variable SYSPBUFF is created at macro invocation time, but the local symbol table is empty, CALL SYMPUT behaves as if the local symbol table was not empty, and creates a local macro variable.

You might find it helpful to use the %PUT statement with the _USER_ option to determine what symbol table the CALL SYMPUT routine has created the variable in.

Example Using CALL SYMPUT with Complete DATA Step and a Nonempty Local Symbol Table

Consider the following example, which contains a complete DATA step with a CALL SYMPUT statement inside a macro:

```
%macro env1(param1);
   data _null_;
      x = 'a token';
      call symput('myvar1',x);
   run;
%mend env1;

%env1(10)

data temp;
   y = "&myvar1";
run;
```

When you submit these statements, you receive an error message:

```
WARNING:  Apparent symbolic reference MYVAR1 not resolved.
```

This message appears for the following reasons:

- the DATA step is complete within the environment of ENV1 (that is, the RUN statement is within the macro)
- the local symbol table of ENV1 is not empty (it contains parameter PARAM1)

Therefore, the CALL SYMPUT routine creates MYVAR1 as a local variable for ENV1, and the value is not available to the subsequent DATA step, which expects a global macro variable.

To see the scopes, add a %PUT statement with the _USER_ option to the macro, and a similar statement in open code. Now invoke the macro as before:

```
%macro env1(param1);
   data _null_;
      x = 'a token';
      call symput('myvar1',x);
   run;

   %put ** Inside the macro: **;
   %put _user_;
%mend env1;

%env1(10)

%put ** In open code: **;
%put _user_;
```

```
data temp;
   y = "&myvar1";  /* ERROR - MYVAR1 is not available in open code. */
run;
```

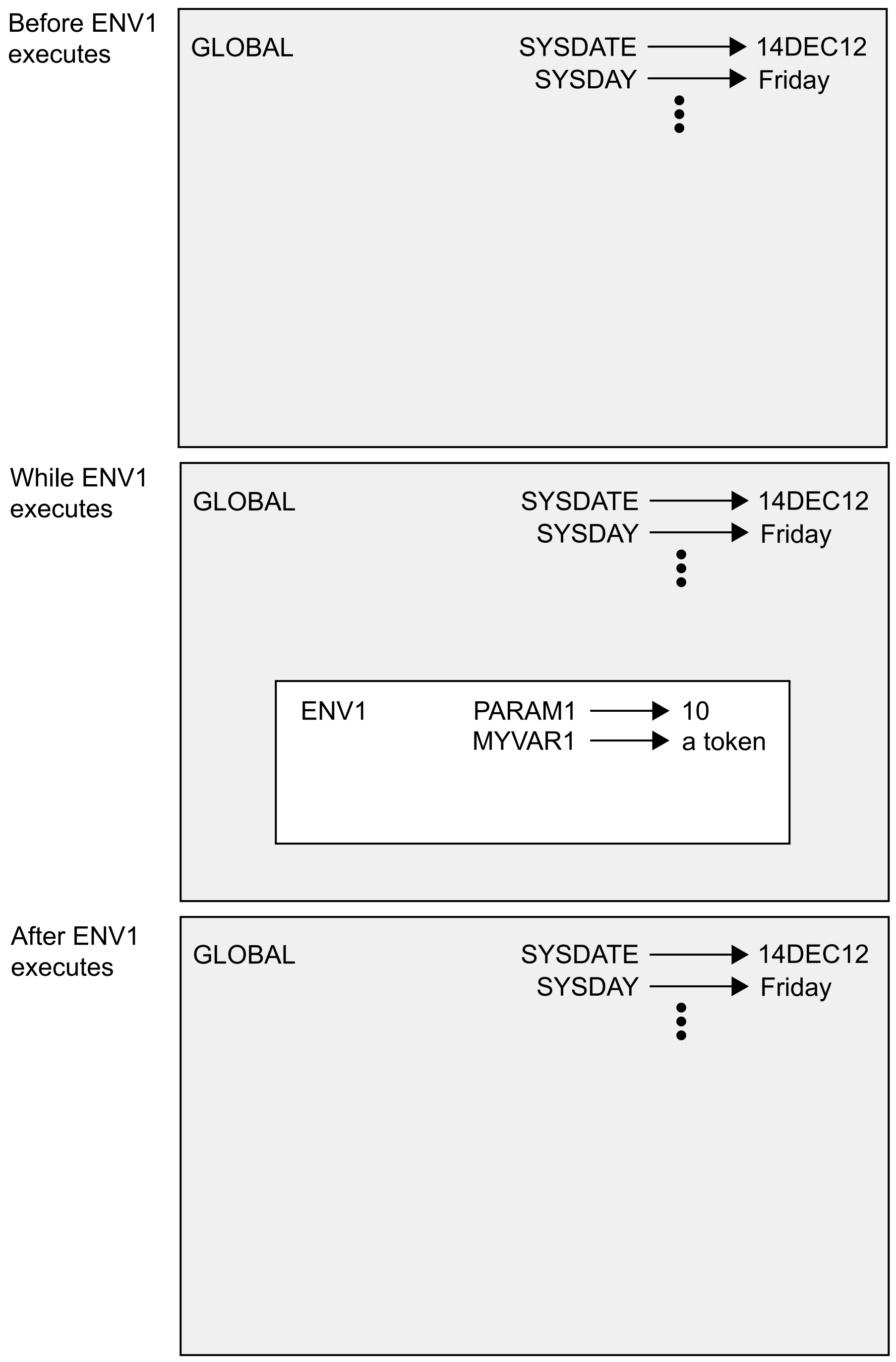
When the %PUT _USER_ statements execute, they write the following information to the SAS log:

```
** Inside the macro: **
ENV1    MYVAR1    a token
ENV1    PARAM1    10

** In open code: **
```

The MYVAR1 macro variable is created by CALL SYMPUT in the local ENV1 symbol table. The %PUT _USER_ statement in open code writes nothing to the SAS log, because no global macro variables are created.

The following figure shows all of the symbol tables in this example.

Figure 5.9 The Symbol Tables with the CALL SYMPUT Routine Generating a Complete DATA Step



Example Using CALL SYMPUT with an Incomplete DATA Step

In the macro ENV2, shown here, the DATA step is not complete within the macro because there is no RUN statement:

```
%macro env2(param2);
   data _null_;
      x = 'a token';
      call symput('myvar2',x);
%mend env2;

%env2(20)
run;

data temp;
   y="&myvar2";
run;
```

These statements execute without errors. The DATA step is complete only when SAS encounters the RUN statement (in this case, in open code). Thus, the current scope of the DATA step is the global scope. CALL SYMPUT creates MYVAR2 as a global macro variable, and the value is available to the subsequent DATA step.

Again, use the %PUT statement with the _USER_ option to illustrate the scopes:

```
%macro env2(param2);
   data _null_;
      x = 'a token';
      call symput('myvar2',x);

   %put ** Inside the macro: **;
   %put _user_;
%mend env2;

%env2(20)

run;

%put ** In open code: **;
%put _user_;

data temp;
   y="&myvar2";
run;
```

When the %PUT _USER_ statement within ENV2 executes, it writes the following to the SAS log:

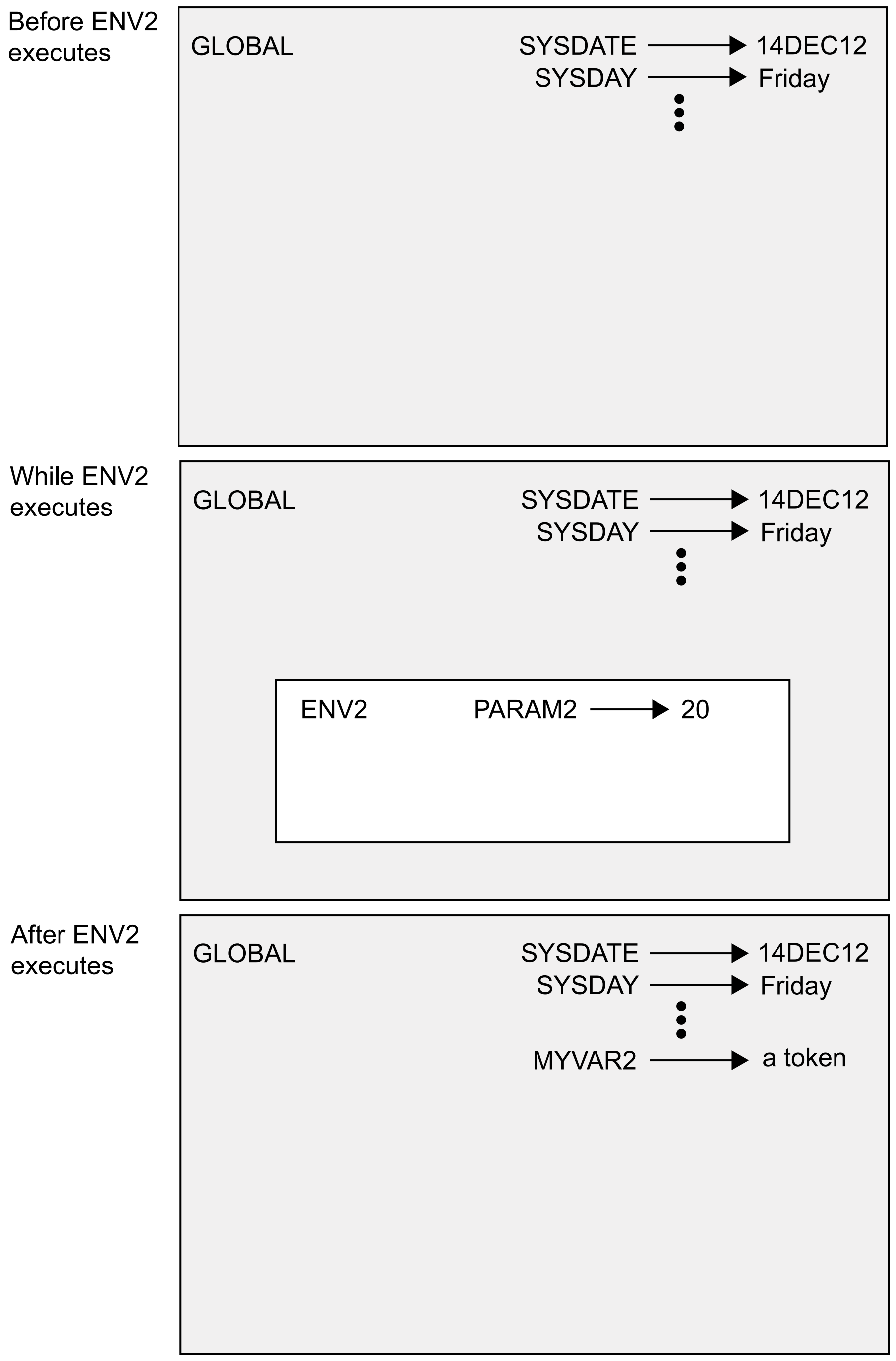
```
** Inside the macro: **
ENV2   PARAM2   20
```

The %PUT _USER_ statement in open code writes the following to the SAS log:

```
** In open code: **
GLOBAL   MYVAR2   a token
```

The following figure shows all the scopes in this example.

Figure 5.10 The Symbol Tables with the CALL SYMPUT Routine Generating an Incomplete DATA Step



Example Using CALL SYMPUT with a Complete DATA Step and an Empty Local Symbol Table

In the following example, ENV3 does not use macro parameters. Therefore, its local symbol table is empty:

```
%macro env3;
   data _null_;
      x = 'a token';
      call symput('myvar3',x);
   run;

   %put ** Inside the macro: **;
   %put _user_;
%mend env3;

%env3

%put ** In open code: **;
%put _user_;

data temp;
   y="&myvar3";
run;
```

In this case, the DATA step is complete and executes within the macro, but the local symbol table is empty. So, CALL SYMPUT creates MYVAR3 in the closest available nonempty symbol table—the global symbol table. Both %PUT statements show that MYVAR3 exists in the global symbol table:

```
** Inside the macro: **
GLOBAL    MYVAR3   a token

** In open code: **
GLOBAL    MYVAR3   a token
```

Example Using CALL SYMPUT with SYSPBUFF and an Empty Local Symbol Table

In the following example, the presence of the SYSPBUFF automatic macro variable causes CALL SYMPUT to behave as if the local symbol table were not empty, even though the macro ENV4 has no parameters or local macro variables:

```
%macro env4 /parmbuff;
 data _null_;
      x = 'a token';
      call symput('myvar4',x);
```

```
   run;

   %put ** Inside the macro: **;
   %put _user_;
   %put &syspbuff;
%mend env4;

%env4

%put ** In open code: **;
%put _user_;
%put &syspbuff;

data temp;
   y="&myvar4";  /* ERROR - MYVAR4 is not available in open code */
run;
```

The presence of the /PARMBUFF specification causes the SYSPBUFF automatic macro variable to be created. So, when you call macro ENV4, CALL SYMPUT creates the macro variable MYVAR4 in the local symbol table (that is, in ENV4's). This action happens even though the macro ENV4 has no parameters and no local variables.

The results of the %PUT statements prove the following:

- the score of MYVAR4 is listed as ENV4
- the reference to SYSPBUFF does not resolve in the open code %PUT statement because SYSPBUFF is local to ENV4

```
** Inside the macro: **
b ENV4     MYVAR4    a token

** In open code: **
WARNING: Apparent symbolic reference SYSPBUFF not resolved.
```

For more information, see “SYSPBUFF Automatic Macro Variable” on page 265.

6

Macro Expressions

Macro Expressions

There are three types of macro expressions: text, logical, and arithmetic. A *text expression* is any combination of text, macro variables, macro functions, or macro calls. Text expressions are resolved to generate text. Here are some examples of text expressions:

- &BEGIN
- %GETLINE
- &PREFIX.PART&SUFFIX
- %UPCASE(&ANSWER)

Logical expressions and *arithmetic expressions* are sequences of operators and operands forming sets of instructions that are evaluated to produce a result. An arithmetic expression contains an arithmetic operator. A logical expression contains a logical operator. The following table shows examples of simple arithmetic and logical expressions:

Table 6.1 Arithmetic and Logical Expressions

Arithmetic Expressions	Logical Expressions
1 + 2	&DAY = FRIDAY
4 * 3	A < a
4 / 2	1 < &INDEX
00FFx - 003Ax	&START NE &END

Defining Arithmetic and Logical Expressions

Evaluating Arithmetic and Logical Expressions

You can use arithmetic and logical expressions in specific macro functions and statements. (See the following table.) The arithmetic and logical expressions in these functions and statements enable you to control the text generated by a macro when it is executed.

Table 6.2 Macro Language Elements That Evaluate Arithmetic and Logical Expressions

%DO *macro-variable=expression* %TO *expression* <%BY *expression*>;
%DO %UNTIL(*expression*);
%DO %WHILE(*expression*);
%EVAL (*expression*);
%IF *expression* %THEN *statement*;
%QSCAN(*argument,expression*<,*delimiters*>)
%QSUBSTR(*argument,expression*<,expression>)
%SCAN(*argument,expression*,<delimiters>)

%SUBSTR(*argument,expression*<,expression>)
%SYSEVALF(*expression,conversion-type*)

You can use text expressions to generate partial or complete arithmetic or logical expressions. The macro processor resolves text expressions before it evaluates the arithmetic or logical expressions. For example, when you submit the following statements, the macro processor resolves the macro variables &A, &B, and &OPERATOR in the %EVAL function, before it evaluates the expression 2 + 5:

```
%let A=2;
%let B=5;
%let operator=+;
%put The result of &A &operator &B is %eval(&A &operator
&B).;
```

When you submit these statements, the %PUT statement writes the following to the log:

```
The result of 2 + 5 is 7.
```

Operands and Operators

Operands in arithmetic or logical expressions are always text. However, an operand that represents a number can be temporarily converted to a numeric value when an expression is evaluated. By default, the macro processor uses integer arithmetic, and only integers and hexadecimal values that represent integers can be converted to a numeric value. Operands that contain a period character (for example 1.0) are not converted. The exception is the %SYSEVALF function. It interprets a period character in its argument as a decimal point and converts the operand to a floating-point value on your operating system.

Note: The values of numeric expressions are restricted to the range of –2**64 to 2**64–1.

Operators in macro expressions are a subset of the operators in the DATA step (Table 6.3 on page 88). However, in the macro language, there is no MAX or MIN operator, and it does not recognize ':', as does the DATA step. The order in which operations are performed when an expression is evaluated is the same in the macro language as in the DATA step. Operations within parentheses are performed first.

Note: Expressions in which comparison operators surround a macro expression, as in 10<&X<20, might be the equivalent of a DATA step compound expression (depending on the expression resolution). To be safe, specify the connecting operator, as in the expression 10<&X AND &X<20.

Note: Datetime constants are internally converted using the BEST12. format.

Table 6.3 Macro Language Operators

Operator	Mnemonic	Precedence	Definition	Example
**		1	exponentiation	2**4
+		2	positive prefix	+(A+B)
-		2	negative prefix	-(A+B)
¬^~	NOT	3	logical not*	NOT A
*		4	multiplication	A*B
/		4	division	A/B
+		5	addition	A+B
-		5	subtraction	A-B
<	LT	6	less than	A<B
<=	LE	6	less than or equal	A<=B
=	EQ	6	equal	A=B
#	IN	6	equal to one of a list**	A#B C D E
¬= ^= ~=	NE	6	not equal*	A NE B
>	GT	6	greater than	A>B
>=	GE	6	greater than or equal	A>=B
&	AND	7	logical and	A=B & C=D
|	OR	8	logical or	A=B | C=D

*The symbol to use depends on your keyboard.

** The default delimiter for list elements is a blank. For more information, see "MINDELIMITER= System Option" on page 455.

** Before using the IN (#) operator, see "MINOPERATOR System Option" on page 457.

** When you use the IN operator, both operands must contain a value. If the operand contains a null value, an error is generated.

CAUTION
Integer expressions that contain exponential, multiplication, or division operators and that use or compute values outside the range -9,007,199,254,740,992 to 9,007,199,254,740,992 might get inexact results.

How the Macro Processor Evaluates Arithmetic Expressions

Evaluating Numeric Operands

The macro facility is a string handling facility. However, in specific situations, the macro processor can evaluate operands that represent numbers as numeric values. The macro processor evaluates an expression that contains an arithmetic operator and operands that represent numbers. Then, it temporarily converts the operands to numeric values and performs the integer arithmetic operation. The result of the evaluation is text.

By default, arithmetic evaluation in most macro statements and functions is performed with integer arithmetic. The exception is the %SYSEVALF function. For more information, see "Evaluating Floating-Point Operands" on page 90. The following macro statements illustrate integer arithmetic evaluation:

```
%let a=%eval(1+2);
%let b=%eval(10*3);
%let c=%eval(4/2);
%let i=%eval(5/3);
%put The value of a is &a;
%put The value of b is &b;
%put The value of c is &c;
%put The value of I is &i;
```

When you submit these statements, the following messages appear in the log:

```
The value of a is 3
The value of b is 30
The value of c is 2
The value of I is 1
```

Notice the result of the last statement. If you perform division on integers that would ordinarily result in a fraction, integer arithmetic discards the fractional part.

When the macro processor evaluates an integer arithmetic expression that contains a character operand, it generates an error. Only operands that contain characters that represent integers or hexadecimal values are converted to numeric values. The following statement shows an incorrect usage:

```
%let d=%eval(10.0+20.0);   /*INCORRECT*/
```

The %EVAL function supports only integer arithmetic. The macro processor does not convert a value containing a period character to a number, and the operands are evaluated as character operands. This statement produces the following error message:

```
ERROR: A character operand was found in the %EVAL function or %IF
condition where a numeric operand is required. The condition was:
10.0+20.0
```

Evaluating Floating-Point Operands

The %SYSEVALF function evaluates arithmetic expressions with operands that represent floating-point values. For example, the following expressions in the %SYSEVALF function are evaluated using floating-point arithmetic:

```
%let a=%sysevalf(10.0*3.0);
%let b=%sysevalf(10.5+20.8);
%let c=%sysevalf(5/3);
%put 10.0*3.0 = &a;
%put 10.5+20.8 = &b;
%put 5/3 = &c;
```

The %PUT statements display the following messages in the log:

```
10.0*3.0 = 30
10.5+20.8 = 31.3
5/3 = 1.6666666667
```

When the %SYSEVALF function evaluates arithmetic expressions, it temporarily converts the operands that represent numbers to floating-point values. The result of the evaluation can represent a floating-point value, but as in integer arithmetic expressions, the result is always text.

The %SYSEVALF function provides conversion type specifications: BOOLEAN, INTEGER, CEIL, and FLOOR. For example, the following %PUT statements return 1, 2, 3, and 2 respectively:

```
%let a=2.5;
%put %sysevalf(&a,boolean);
%put %sysevalf(&a,integer);
%put %sysevalf(&a,ceil);
%put %sysevalf(&a,floor);
```

These conversion types modify the value returned by %SYSEVALF so that it can be used in other macro expressions that require integer or Boolean values.

CAUTION
Specify a conversion type for the %SYSEVALF function. If you use the %SYSEVALF function in macro expressions or assign its results to macro variables that are used in other macro expressions, then errors or unexpected results might occur if the %SYSEVALF function returns missing or floating-point values. To prevent errors, specify a conversion type that returns a value compatible with other macro expressions. For more information about using conversion types, see "%SYSEVALF Macro Function" on page 352.

How the Macro Processor Evaluates Logical Expressions

Comparing Numeric Operands in Logical Expressions

A logical, or Boolean, expression returns a value that is evaluated as true or false. In the macro language, any numeric value other than 0 is true and a value of 0 is false.

When the macro processor evaluates logical expressions that contain operands that represent numbers, it converts the characters temporarily to numeric values. To illustrate how the macro processor evaluates logical expressions with numeric operands, consider the following macro definition:

```
%macro compnum(first,second);
   %if &first>&second %then %put &first is greater than &second;
   %else %if &first=&second %then %put &first equals &second;
   %else %put &first is less than &second;
%mend compnum;
```

Invoke the COMPNUM macro with these values:

```
%compnum(1,2)
%compnum(-1,0)
```

The following results are displayed in the log:

```
1 is less than 2
-1 is less than 0
```

The results show that the operands in the logical expressions were evaluated as numeric values.

Comparing Floating-Point or Missing Values

You must use the %SYSEVALF function to evaluate logical expressions containing floating-point or missing values. To illustrate comparisons with floating-point and missing values, consider the following macro that compares parameters passed to it with the %SYSEVALF function and places the result in the log:

```
%macro compflt(first,second);
   %if %sysevalf(&first>&second) %then %put &first is greater than
&second;
```

```
   %else %if  %sysevalf(&first=&second) %then %put &first equals
&second;
   %else %put &first is less than &second;
%mend compflt;
```

Invoke the COMPFLT macro with these values:

```
%compflt (1.2,.9)
%compflt (-.1,.)
%compflt (0,.)
```

The following values are written in the log:

```
1.2 is greater than .9
-.1 is greater than .
0 is greater than .
```

The results show that the %SYSEVALF function evaluated the floating-point and missing values.

Comparing Character Operands in Logical Expressions

To illustrate how the macro processor evaluates logical expressions, consider the COMPCHAR macro. Invoking the COMPCHAR macro compares the values passed as parameters and places the result in the log.

```
%macro compchar(first,second);
   %if &first>&second %then %put &first comes after &second;
   %else %put &first comes before &second;
%mend compchar;
```

Invoke the macro COMPCHAR with these values:

```
%compchar(a,b)
%compchar(.,1)
%compchar(Z,E)
```

The following results are printed in the log:

```
a comes before b
. comes before 1
Z comes after E
```

When the macro processor evaluates expressions with character operands, it uses the sort sequence of the host operating system for the comparison. The comparisons in these examples work with both EBCDIC and ASCII sort sequences.

A special case of a character operand is an operand that looks numeric but contains a period character. If you use an operand with a period character in an expression, both operands are compared as character values. This can lead to unexpected results. So that you can understand and better anticipate results, look at the following examples.

Invoke the COMPNUM macro with these values:

```
%compnum(10,2.0)
```

The following values are written to the log:

```
10 is less than 2.0
```

Because the %IF-THEN statement in the COMPNUM macro uses integer evaluation, it does not convert the operands with decimal points to numeric values. The operands are compared as character strings using the host sort sequence, which is the comparison of characters with smallest-to-largest values. For example, lowercase letters might have smaller values than uppercase, and uppercase letters might have smaller values than digits.

CAUTION
The host sort sequence determines comparison results. If you use a macro definition on more than one operating system, comparison results might differ because the sort sequence of one host operating system might differ from the other system. For more information about host sort sequences, see "SORT Procedure" in *Base SAS Procedures Guide*.

7

Macro Quoting

Macro Quoting

Masking Special Characters and Mnemonics

The macro language is a character-based language. Even variables that appear to be numeric are generally treated as character variables (except during expression evaluation). Therefore, the macro processor enables you to generate all sorts of special characters as text. But because the macro language includes some of the same special characters, an ambiguity often arises. The macro processor must know whether to interpret a particular special character (for example, a semicolon or % sign) or a mnemonic (for example, GE or AND) as text or as a symbol in the macro language. Macro quoting functions resolve these ambiguities by masking the significance of special characters so that the macro processor does not misinterpret them.

The following special characters and mnemonics might require masking when they appear in text strings:

Table 7.1 *Special Characters and Mnemonics*

blank	)	=	LT
;	(	|	GE
¬	+	AND	GT
^	—	OR	IN
~	*	NOT	%
, (comma)	/	EQ	&
'	<	NE	#
"	>	LE	

Understanding Why Macro Quoting Is Necessary

Macro quoting functions tell the macro processor to interpret special characters and mnemonics as text rather than as part of the macro language. If you did not use a macro quoting function to mask the special characters, the macro processor or the rest of SAS might give the character a meaning that you did not intend. Here are some examples of the types of ambiguities that can arise when text strings contain special characters and mnemonics:

- Is `%sign` a call to the macro SIGN or a phrase “percent sign”?
- Is OR the mnemonic Boolean operator or the abbreviation for Oregon?
- Is the quotation mark in O’Malley an unbalanced single quotation mark or just part of the name?
- Is Boys&Girls a reference to the macro variable &GIRLS or a group of children?
- Is GE the mnemonic for “greater than or equal” or is it short for General Electric?
- Which statement does a semicolon end?
- Does a comma separate parameters, or is it part of the value of one of the parameters?

Macro quoting functions enable you to clearly indicate to the macro processor how it is to interpret special characters and mnemonics.

Here is an example, using the simplest macro quoting function, %STR. Suppose you want to assign a PROC PRINT statement and a RUN statement to the macro variable PRINT. Here is the erroneous statement:

```
%let print=proc print; run;;  /* undesirable results */
```

This code is ambiguous. Are the semicolons that follow PRINT and RUN part of the value of the macro variable PRINT, or does one of them end the %LET statement? If you do not tell the macro processor what to do, it interprets the semicolon after PRINT as the end of the %LET statement. So the value of the PRINT macro variable would be the following:

```
proc print
```

The rest of the characters (RUN;;) would be simply the next part of the program.

To avoid the ambiguity and correctly assign the value of PRINT, you must mask the semicolons with the macro quoting function %STR, as follows:

```
   %let print=%str(proc print; run;);
```

Overview of Macro Quoting Functions

The following macro quoting functions are most commonly used:

- %STR and %NRSTR

- %BQUOTE and %NRBQUOTE
- %SUPERQ

For the paired macro quoting functions, the function beginning with NR affects the same category of special characters that are masked by the plain macro quoting function as well as ampersands and percent signs. In effect, the NR functions prevent macro and macro variable resolution. To help you remember which does which, try associating the NR in the macro quoting function names with the words “not resolved” — that is, macros and macro variables are not resolved when you use these functions.

The macro quoting functions with B in their names are useful for macro quoting unmatched quotation marks and parentheses. To help you remember the B, try associating B with “by itself”.

The %SUPERQ macro quoting function is unlike the other macro quoting functions in that it does not have a mate and works differently. For more information, see “%SUPERQ Macro Function” on page 347.

The macro quoting functions can also be divided into two types, depending on when they take effect:

compilation functions
: cause the macro processor to interpret special characters as text in a macro program statement in open code or while compiling (constructing) a macro. The %STR and %NRSTR functions are compilation functions. For more information, see “%STR Macro Function” on page 342.

execution functions
: cause the macro processor to treat special characters that result from resolving a macro expression as text (such as a macro variable reference, a macro invocation, or the argument of an %EVAL function). They are called execution functions because resolution occurs during macro execution or during execution of a macro program statement in open code. The macro processor resolves the expression as far as possible, issues any warning messages for macro variable references or macro invocations that it cannot resolve, and quotes the result. The %BQUOTE and %NRBQUOTE functions are execution functions. For more information, see “%BQUOTE Macro Function” on page 314.

The %SUPERQ function takes as its argument a macro variable name (or a macro expression that yields a macro variable name). The argument must not be a reference to the macro variable whose value you are masking. That is, do not include the & before the name.

Note: Two other execution macro quoting functions exist: %QUOTE and %NRQUOTE. They are useful for unique macro quoting needs and for compatibility with older macro applications. For more information, see “%QUOTE Macro Function” on page 334.

Passing Parameters That Contain Special Characters and Mnemonics

Using an execution macro quoting function in the macro definition is the simplest and best way for the macro processor to accept resolved values that might contain special characters. However, if you discover that you need to pass parameter values such as `or` when a macro has not been defined with an execution macro quoting function, you can do so by masking the value in the macro invocation. The logic of the process is as follows:

1. When you mask a special character with a macro quoting function, it remains masked as long as it is within the macro facility (unless you use the "%UNQUOTE Macro Function" on page 370).
2. The macro processor constructs the complete macro invocation before beginning to execute the macro.
3. Therefore, you can mask the value in the invocation with the %STR function. The masking is not needed when the macro processor is constructing the invocation. The value is already masked by a macro quoting function when macro execution begins and therefore does not cause problems during macro execution.

For example, suppose a macro named ORDERX does not use the %BQUOTE function. You can pass the value `or` to the ORDERX macro with the following invocation:

```
%orderx(%str(or))
```

However, placing the macro quoting function in the macro definition makes the macro much easier for you to invoke.

Deciding When to Use a Macro Quoting Function and Which Function to Use

Use a macro quoting function anytime you want to assign to a macro variable a special character that could be interpreted as part of the macro language. The following table describes the special characters to mask when used as part of a text string and which macro quoting functions are useful in each situation.

Table 7.2 Special Characters and Macro Quoting Guidelines

Special Character	Must Be Masked	Quoted by All Macro Quoting Functions?	Remarks
+-*/<>=^|¬ ~ # LE LT EQ NE GE GT AND OR NOT IN	To prevent it from being treated as an operator in the argument of an %EVAL function	Yes	AND, OR, IN, and NOT need to be masked because they are interpreted as mnemonic operators by an %EVAL and by %SYSEVALF.
blank	To maintain, rather than ignore, a leading, trailing, or isolated blank	Yes	
;	To prevent a macro program statement from ending prematurely	Yes	
, (comma)	To prevent it from indicating a new function argument, parameter, or parameter value	Yes	
' " ( )	If it might be unmatched	No	Arguments that might contain quotation marks and parentheses should be masked with a macro quoting function so that the macro facility interprets the single and double quotation marks and parentheses as text rather than macro language symbols or possibly unmatched quotation marks or parentheses for the SAS language. With %STR, %NRSTR, %QUOTE, and %NRQUOTE, unmatched quotation marks and parentheses must be marked with a % sign. You do not have to mark unmatched symbols in the arguments of %BQUOTE, %NRBQUOTE, and %SUPERQ.

Special Character	Must Be Masked	Quoted by All Macro Quoting Functions?	Remarks
%*name* &*name*	(Depends on what the expression might resolve to)	No	%NRSTR, %NRBQUOTE, and %NRQUOTE mask these patterns. To use %SUPERQ with a macro variable, omit the ampersand from *name*.

The macro facility allows you as much flexibility as possible in designing your macros. You need to mask a special character with a macro quoting function only when the macro processor would otherwise interpret the special character as part of the macro language rather than as text. For example, in this statement that you must use a macro quoting function to mask the first two semicolons to make them part of the text:

```
%let p=%str(proc print; run;);
```

However, in the macro PR, shown here, you do not need to use a macro quoting function to mask the semicolons after PRINT and RUN:

```
%macro pr(start);
   %if &start=yes %then
      %do;
         %put proc print requested;
         proc print;
         run;
      %end;
%mend pr;
```

Because the macro processor does not expect a semicolon within the %DO group, the semicolons after PRINT and RUN are not ambiguous, and they are interpreted as text.

Although it is not possible to give a series of rules that cover every situation, the following sections describe how to use each macro quoting function. Table 7.6 on page 113 provides a summary of the various characters that might need masking and of which macro quoting function is useful in each situation.

Note: You can also perform the inverse of a macro quoting function — that is, remove the tokenization provided by macro quoting functions. For an example of when the %UNQUOTE function is useful, see "Unquoting Text" on page 115.

%STR and %NRSTR Functions

Using %STR and %NRSTR Functions

If a special character or mnemonic affects how the macro processor constructs macro program statements, you must mask the item during macro compilation (or during the compilation of a macro program statement in open code) by using either the %STR or %NRSTR macro quoting functions.

These macro quoting functions mask the following special characters and mnemonics:

Table 7.3 Special Characters Masked by the %STR and %NRSTR Functions

blank	)	=	NE	
;	(	|	LE	
¬	+	#	LT	
^	—	AND	GE	
~	*	OR	GT	
, (comma)	/	NOT		
'	<	IN		
"	>	EQ		

In addition to these special characters and mnemonics, %NRSTR masks & and %.

Note: If an unmatched single or double quotation mark or an open or close parenthesis is used with %STR or %NRSTR, these characters must be preceded by a percent sign (%).

When you use %STR or %NRSTR, the macro processor does not receive these functions and their arguments when it executes a macro. It receives only the results of these functions because these functions work when a macro compiles. By the time the macro executes, the string is already masked by a macro quoting function. Therefore, %STR and %NRSTR are useful for masking strings that are constants, such as sections of SAS code. In particular, %NRSTR is a good choice for masking strings that contain % and & signs. However, these functions are not so useful for masking strings that contain references to macro variables because it is possible that the macro variable could resolve to a value not quotable by %STR or %NRSTR. For example, the string could contain an unmarked, unmatched open parenthesis.

Using Unmatched Quotation Marks and Parentheses with %STR and %NRSTR

If the argument to %STR or %NRSTR contains an unmatched single or double quotation mark or an unmatched open or close parenthesis, precede each of these characters with a % sign. The following table shows some examples of this technique.

Table 7.4 *Examples of Marking Unmatched Quotation Marks and Parentheses with %STR and %NRSTR*

Notation	Description	Example	Quoted Value Stored
%'	unmatched single quotation mark	`%let myvar= %str(a%');`	`a'`
%"	unmatched double quotation mark	`%let myvar= %str(title %"first);`	`title "first`
%(	unmatched open parenthesis	`%let myvar=%str (log%(12);`	`log(12`
%)	unmatched close parenthesis	`%let myvar=%str (345%));`	`345)`

Using % Signs with %STR

In general, if you want to mask a % sign with a macro quoting function at compilation, use %NRSTR. There is one case where you can use %STR to mask a % sign: when the % sign does not have any text following it that could be construed by the macro processor as a macro name. The % sign must be marked by another % sign. Here are some examples.

Table 7.5 Examples of Masking % Signs with %STR

Notation	Description	Example	Quoted Value Stored
'%'	% sign before a matched single quotation mark	`%let myvar= %str('%');`	'%'
%%%'	% sign before an unmatched single quotation mark	`%let myvar= %str(%%%');`	`%'`
""%%	% sign after a matched double quotation mark	`%let myvar= %str(""%%);`	`""%`
%%%%	two % signs in a row	`%let myvar= %str(%%%%);`	`%%`

There is an issue that can occur when a macro variable value ends with % and, at resolution, is followed immediately by a special character and another macro variable. In that case, the special character and the second macro variable value might print twice. The following example illustrates this issue:

```
%let var1=ABC%;
%let var2=DEF;
%put &var1.#&var2.;
```

Here is the result:

```
ABC%#DEF#DEF
```

To resolve the issue in this case, use the %NRSTR function and two percent signs in the VAR1 macro variable assignment, as shown in the following example:

```
%let var1=%nrstr(ABC%%);
%let var2=DEF;
%put &var1.#&var2.;
```

This masks the % in the VAR1 macro variable value, which resolves the issue. Here is the result:

```
ABC%#DEF
```

Note: The %STR function does not mask % and, as a result, does not resolve this issue.

Examples Using %STR

The %STR function in the following %LET statement prevents the semicolon after PROC PRINT from being interpreted as the ending semicolon for the %LET statement:

```
%let printit=%str(proc print; run;);
```

As a more complex example, the macro KEEPIT1 shows how the %STR function works in a macro definition:

```
%macro keepit1(size);
   %if &size=big %then %put %str(keep city _numeric_;);
   %else %put %str(keep city;);
%mend keepit1;
```

Call the macro as follows:

```
%keepit1(big)
```

This code produces the following statement:

```
keep city _numeric_;
```

When you use the %STR function in the %IF-%THEN statement, the macro processor interprets the first semicolon after the word %THEN as text. The second semicolon ends the %THEN statement, and the %ELSE statement immediately follows the %THEN statement. Thus, the macro processor compiles the statements as you intended. However, if you omit the %STR function, the macro processor interprets the first semicolon after the word %THEN as the end of the %THEN clause. The next semicolon as constant text. Because only a %THEN clause can precede a %ELSE clause, the semicolon as constant text causes the macro processor to issue an error message and not compile the macro.

In the %ELSE statement, the %STR function causes the macro processor to treat the first semicolon in the statement as text and the second one as the end of the %ELSE clause. Therefore, the semicolon that ends the KEEP statement is part of the conditional execution. If you omit the %STR function, the first semicolon ends the %ELSE clause and the second semicolon is outside the conditional execution. It is generated as text each time the macro executes. (In this example, the placement of the semicolon does not affect the SAS code.) Again, using %STR causes the macro KEEPIT1 to compile as you intended.

Here is an example that uses %STR to mask a string that contains an unmatched single quotation mark. Note the use of the % sign before the quotation mark:

```
%let innocent=%str(I didn%'t do it!);
```

Examples Using %NRSTR

Suppose you want the name (not the value) of a macro variable to be printed by the %PUT statement. To do so, you must use the %NRSTR function to mask the & and prevent the resolution of the macro variable, as in the following example:

```
%macro example;
   %local myvar;
   %let myvar=abc;
   %put %nrstr(The string &myvar appears in log output,);
   %put instead of the variable value.;
%mend example;

%example
```

This code writes the following text to the SAS log:

```
The string &myvar appears in log output,
instead of the variable value.
```

If you did not use the %NRSTR function or if you used %STR, the following undesired output would appear in the SAS log:

```
The string abc appears in log output,
instead of the variable value.
```

The %NRSTR function prevents the & from triggering macro variable resolution.

The %NRSTR function is also useful when the macro definition contains patterns that the macro processor would ordinarily recognize as macro variable references, as in the following program:

```
%macro credits(d=%nrstr(Mary&Stacy&Joan Ltd.));
   footnote "Designed by &d";
%mend credits;
```

Using %NRSTR causes the macro processor to treat &STACY and &JOAN simply as part of the text in the value of D; the macro processor does not issue warning messages for unresolvable macro variable references. Suppose you invoke the macro CREDITS with the default value of D, as follows:

```
%credits()
```

Submitting this program generates the following FOOTNOTE statement:

```
footnote "Designed by Mary&Stacy&Joan Ltd.";
```

If you omit the %NRSTR function, the macro processor attempts to resolve the references &STACY and &JOAN as part of the resolution of &D in the FOOTNOTE statement. The macro processor issues these warning messages (assuming the SERROR system option, described in “System Options for Macros” on page 434 is active) because no such macro variables exist:

```
WARNING: Apparent symbolic reference STACY not resolved.
WARNING: Apparent symbolic reference JOAN not resolved.
```

Here is a final example of using %NRSTR. Suppose you wanted a text string to include the name of a macro function: `This is the result of %NRSTR`. Here is the program:

```
%put This is the result of %nrstr(%nrstr);
```

You must use %NRSTR to mask the % sign at compilation, so the macro processor does not try to invoke %NRSTR a second time. If you did not use %NRSTR to mask the string `%nrstr`, the macro processor would complain about a missing open parenthesis for the function.

%BQUOTE and %NRBQUOTE Functions

Using %BQUOTE and %NRBQUOTE Functions

%BQUOTE and %NRBQUOTE mask values during execution of a macro or a macro language statement in open code. These functions instruct the macro processor to resolve a macro expression as far as possible and mask the result, issuing any warning messages for macro variable references or macro invocations that it cannot resolve. These functions mask all the characters that %STR and %NRSTR mask with the addition of unmarked percent signs; unmatched, unmarked single and double quotation marks; and unmatched, unmarked opening and closing parentheses. That means that you do not have to precede an unmatched quotation mark with a % sign, as you must when using %STR and %NRSTR.

The %BQUOTE function treats all parentheses and quotation marks produced by resolving macro variable references or macro calls as special characters to be masked at execution time. (It does not mask parentheses or quotation marks that are in the argument at compile time.) Therefore, it does not matter whether quotation marks and parentheses in the resolved value are matched; each one is masked individually.

The %NRBQUOTE function is useful when you want a value to be resolved when first encountered, if possible, but you do not want any ampersands or percent signs in the result to be interpreted as operators by an %EVAL function.

If the argument of the %NRBQUOTE function contains an unresolvable macro variable reference or macro invocation, the macro processor issues a warning message before it masks the ampersand or percent sign (assuming the SERROR or MERROR system option, described in “System Options for Macros” on page 434 is in effect). To suppress the message for unresolved macro variables, use the %SUPERQ function (discussed later in this section) instead.

The %BQUOTE and %NRBQUOTE functions operate during execution and are more flexible than %STR and %NRSTR.

Examples Using %BQUOTE

In the following code, the %IF-%THEN statement in %TEST uses %BQUOTE to prevent an error if the macro variable STATE resolves to OR (for Oregon), which the macro processor would interpret as the logical operator OR otherwise:

```
%macro test(state);
   %if %bquote(&state)=%str(OR) %then %put Oregon Dept. of Revenue;
%mend test;

%test(OR);
```

Note: This example works if you use %STR, but it is not robust or good programming practice. Because you cannot guarantee what &STATE is going to resolve to, you need to use %BQUOTE to mask the resolution of the macro variable at execution time, not the name of the variable itself at compile time.

In the following example, a DATA step creates a character value containing a single quotation mark and assigns that value to a macro variable. The macro READIT then uses the %BQUOTE function to enable a %IF condition to accept the unmatched single quotation mark:

```
data test;
   store="Susan's Office Supplies";
   call symput('s',store);
run;

%macro readit;
   %if %bquote(&s) ne %then %put *** valid ***;
   %else %put *** null value ***;
%mend readit;

%readit
```

When you assign the value `Susan's Office Supplies` to STORE in the DATA step, enclosing the character string in double quotation marks enables you to use an unmatched single quotation mark in the string. SAS stores the value of STORE:

```
Susan's Office Supplies
```

The CALL SYMPUT routine assigns that value (containing an unmatched single quotation mark) as the value of the macro variable S. If you do not use the %BQUOTE function when you reference S in the macro READIT, the macro processor issues an error message for an invalid operand in the %IF condition.

When you submit the code, the following is written to the SAS log:

```
*** valid ***
```

Referring to Already Quoted Variables

Items that have been masked by a macro quoting function, such as the value of WHOSE in the following program, remain masked as long as the item is being used by the macro processor. When you use the value of WHOSE later in a macro program statement, you do not need to mask the reference again.

```
/* Use %STR to mask the constant, and use a  % sign to mark */
/* the unmatched single quotation mark. */
%let whose=%str(John%'s);

/* You don't need to mask the macro reference, because it was */
/* masked in the %LET statement, and remains masked. */
%put *** This coat is &whose ***;
```

Here is the output from the %PUT statement that is written to the SAS log:

```
*** This coat is John's ***
```

Deciding How Much Text to Mask with a Macro Quoting Function

In each of the following statements, the macro processor treats the masked semicolons as text:

```
%let p=%str(proc print; run;);
%let p=proc %str(print;) %str(run;);
%let p=proc print%str(;) run%str(;);
```

The value of P is the same in each case:

```
proc print; run;
```

The results of the three %LET statements are the same because when you mask text with a macro quoting function, the macro processor quotes only the items that the function recognizes. Other text enclosed in the function remains unchanged. Therefore, the third %LET statement is the minimalist approach to macro quoting. However, masking large blocks of text with a macro quoting function is harmless and actually results in code that is much easier to read (such as the first %LET statement).

%SUPERQ Function

Using %SUPERQ

The %SUPERQ function locates the macro variable named in its argument and quotes the value of that macro variable without permitting any resolution to occur. It masks all items that might require macro quoting at macro execution. Because %SUPERQ does not attempt any resolution of its argument, the macro processor does not issue any warning messages that a macro variable reference or a macro invocation has not been resolved. Therefore, even when the %NRBQUOTE function enables the program to work correctly, you can use the %SUPERQ function to eliminate unwanted warning messages from the SAS log. %SUPERQ takes as its argument either a macro variable name without an ampersand or a text expression that yields a macro variable name.

%SUPERQ retrieves the value of a macro variable from the macro symbol table and quotes it immediately, preventing the macro processor from making any attempt to resolve anything that might occur in the resolved value. For example, if the macro variable CORPNAME resolves to `Smith&Jones`, using %SUPERQ prevents the macro processor from attempting to further resolve `&Jones`. This %LET statement successfully assigns the value `Smith&Jones` to TESTVAR:

```
%let testvar=%superq(corpname);
```

Examples of Using %SUPERQ

This example shows how the %SUPERQ function affects two macro invocations, one for a macro that has been defined and one for an undefined macro. Here is the definition for macro A.

```
%macro a;
   %put *** This is macro a. ***;
%mend a;
```

Suppose that you have a DATA step that puts the string %A %X into global macro variable VAL. Here is a simple example:

```
data input;
   input val $1-15;
   call symput("val", strip(val));
datalines;
%A %X
;
run;
```

If you use %PUT to display the value of VAL, the macro processor attempts to execute macro %A and %X as shown in the following example:

```
%put *** &val ***;
```

```
*** This is macro a. ***
WARNING: Apparent invocation of macro X not resolved.
*** %X ***
```

To prevent the macro processor from executing %A and %X, use %SUPERQ:

```
%put *** %superq(val) ***;  /* Note absence of ampersand */
```

```
*** %A %X ***
```

It does not invoke the macro A, and it does not issue a warning message stating that %X was not resolved.

The following two examples compare the %SUPERQ function with other macro quoting functions.

Using the %SUPERQ Function to Prevent Warning Messages

The sections about the %NRBQUOTE function show that it causes the macro processor to attempt to resolve the patterns *&name* and *%name* the first time it encounters them during macro execution. If the macro processor cannot resolve them, it quotes the ampersand or percent sign so that later uses of the value do not cause the macro processor to recognize them. However, if the MERROR or SERROR option is in effect, the macro processor issues a warning message that the reference or invocation was not resolved.

The macro VALIDATEFIRMS, shown here, shows how the %SUPERQ function can prevent unwanted warning messages:

```
%macro validatefirms;
    %let name=%superq(firmname);
    %if &name ne %then
       %put *** &name is valid ***;
    %else
       %put *** Firm name is missing ***;
%mend validatefirms;
```

This macro validates a firm name that is stored in global macro variable FIRMNAME. Suppose that you have data set Firms, which stores firm names:

```
data firms;
   input firmname $1-20;
datalines;
A&A Autos
Santos&D'Amato
Ann's Coffee Shop
;
run;
```

Here is a DATA step that iterates through the names in data set Firms and uses macro VALIDATEFIRMS to validate each name:

```
data _null_;
   set firmdata;
   call symputx("firmname", strip(firm));
   call execute("%nrstr(%validatefirms)");
run;
%symdel firmname;
```

The CALL SYMPUTX statement assigns the current value of variable FirmName to global macro variable FIRMNAME. The CALL EXECUTE statement invokes macro %VALIDATEFIRMS. When the DATA step is executed, the following is written to the SAS log:

```
*** A&A Autos is valid ***
*** Santos&D'Amato is valid ***
*** Ann's Coffee Shop is valid ***
```

Using the %SUPERQ Function to Ignore Macro Keywords

The %SUPERQ function is useful when working with unknown input that could be interpreted as macro keywords by the macro processor. Suppose that you have a data set that contains message strings that were imported from a user comment application. Here is a simple example:

```
data messages;
   input id msg $4-70;
datalines;
1  There's a 70% chance of rain today.
2  J&L Market has the best deals this week.
3  Type echo %HOMEPATH% to see your home path.
;
run;
```

You can create a macro and another DATA step to print each message in data set MESSAGES. However, in this case, each message requires quoting:

- Message 1 contains an unmatched single quotation mark.
- Message 2 contains &L, which the macro processor attempts to resolve as macro variable L.
- Message 3 contains %HOMEPATH, which the macro processor attempts to resolve as macro HOMEPATH.

The %SUPERQ function can be used to prevent issues when printing these messages as shown in the following example:

```
%macro printMsg;
   %put Message &id: %superq(msg);
%mend printMsg;
```

```
data _null_;
   set messages;
   call symputx("id", strip(id));
   call symputx("msg", strip(msg));
   call execute("%nrstr(%printMsg)");
run;

%symdel msg id;
```

Macro %PRINTMSG prints the message stored in global macro variable MSG. The %SUPERQ function masks special characters and prevents the macro processor from attempting to resolve what appear to be macros and macro variables. The _NULL_ DATA step iterates through the messages in MESSAGES. For each observation:

- The first CALL SYMPUTX statement puts the value of variable ID in global macro variable ID.
- The second CALL SYMPUTX statement puts the value of variable MSG in global macro variable MSG.
- The CALL EXECUTE statement invokes macro %PRINTMSG. The %NRSTR function prevents macro %PRINTMSG from being executed when the DATA step is compiled..

When this code is executed, the following is written to the SAS log:

```
Message 1: There's a 70% chance of rain today.
Message 2: J&L Market has the best deals this week.
Message 3: Type echo %HOMEPATH% to see your home path.
```

No warnings or errors are written to the SAS log.

Summary of Macro Quoting Functions and the Characters That They Mask

Different macro quoting functions mask different special characters and mnemonics so that the macro facility interprets them as text instead of as macro language symbols.

The following table divides the symbols into categories and shows which macro quoting functions mask which symbols.

Table 7.6 Summary of Special Characters and Macro Quoting Functions by Item

Group	Items	Macro Quoting Functions
A	+ — */<>=¬^|~;, # blank AND OR NOT EQ NE LE LT GE GT IN	all

Group	Items	Macro Quoting Functions
B	&%	%NRSTR, %NRBQUOTE, %SUPERQ, %NRQUOTE
C	unmatched' "()	%BQUOTE, %NRBQUOTE, %SUPERQ, %STR*, %NRSTR*, %QUOTE*, %NRQUOTE*

Table 7.7 *By Function*

Function	Affects Groups	Works At
%STR	A, C*	Macro compilation
%NRSTR	A, B, C*	Macro compilation
%BQUOTE	A, C	Macro execution
%NRBQUOTE	A, B, C	Macro execution
%SUPERQ	A, B, C	Macro execution (prevents resolution)
%QUOTE	A, C*	Macro execution. Requires unmatched quotation marks and parentheses to be marked with a percent sign (%).
%NRQUOTE	A, B, C*	Macro execution. Requires unmatched quotation marks and parentheses to be marked with a percent sign (%).
*Unmatched quotation marks and parentheses must be marked with a percent sign (%) when used with %STR, %NRSTR, %QUOTE, and %NRQUOTE.		

Unquoting Text

Restoring the Significance of Symbols

To *unquote* a value means to restore the significance of symbols in an item that was previously masked by a macro quoting function.

Usually, after an item has been masked by a macro quoting function, it retains its special status until one of the following occurs:

- You enclose the item with the %UNQUOTE function. (For more information, see "%UNQUOTE Macro Function" on page 370.)
- The item leaves the word scanner and is passed to the DATA step compiler, SAS procedures, SAS macro facility, or other parts of SAS.
- The item is returned as an unquoted result by the %SCAN, %SUBSTR, or %UPCASE function. (To retain a value's masked status during one of these operations, use the %QSCAN, %QSUBSTR, or %QUPCASE function. For more information, see "Other Functions That Perform Macro Quoting" on page 119.)

As a rule, you do not need to unquote an item because it is automatically unquoted when the item is passed from the word scanner to the rest of SAS. Under two circumstances, however, you might need to use the %UNQUOTE function to restore the original significance to a masked item:

- when you want to use a value with its restored meaning later in the same macro in which its value was previously masked by a macro quoting function
- when masking text with a macro quoting function changes how the word scanner tokenizes it, producing SAS statements that look correct but that the SAS compiler does not recognize

Example of Unquoting

The following example illustrates using a value twice: once in macro quoted form and once in unquoted form. Suppose that you have an application that randomly samples and records readings from a continuous process and stores them in a data set. Here is some sample data.

```
data samples;
   length test $5;
   input sample v1 v2 test $;
datalines;
1 1 17 <=
2 7 4  =>
3 3 1  =
4 9 5  =>
;
run;
```

Each observation contains two sample readings and a logical operator that is used when comparing the two values. To ensure that the process is operating within design specifications, V1 and V2 are compared using the operator in TEST. If the comparison evaluates to True, the process is operating normally. Otherwise, it is operating outside of its design specifications. The following macro performs the test on a specific sample.

```
%macro testsample;
   %let op = %bquote(&test);
   %if &op = %str(=<) %then %let op = %str(<=);
   %else %if &op = %str(=>) %then %let op = %str(>=);
   %if &v1 %unquote(&op) &v2 %then
      %put Sample &sample: V1=&v1, V2=&v2, Test is V1 &op V2: **TEST
PASSED**;
   %else
      %do;
         %put Sample &sample: V1=&v1, V2=&v2, Test is V1 &op V2:
**TEST FAILED**;
      %end;
%mend testsample;
```

Macro variables SAMPLE, V1, V2, and TEST are global macro variables that are set in a DATA step that follows. The value of TEST is quoted using %BQUOTE function, and the result is assigned to macro variable OP. The %BQUOTE function masks resolved items including unmatched, unmarked quotation marks and parentheses (but excluding the ampersand and percent sign).

The %IF condition compares the value of the macro variable OP to a string to see whether the value of OP contains the correct symbols for the operator. If the value contains symbols in the wrong order, the %THEN statement corrects the symbols. Because a value masked by a macro quoting function remains masked, you do not need to mask the reference &OP in the left side of the %IF condition.

Because you can see the characters in the right side of the %IF condition and in the %LET statement when you define the macro, you can use the %STR function to mask them. Masking them once at compilation is more efficient than masking them at each execution of TEST.

To use the value of the macro variable OP as the operator in the %IF condition, the %UNQUOTE function is used to restore the meaning of the operator. The %PUT statements write the result of the test to the SAS log.

The following DATA step executes macro %TESTSAMPLE on each sample in data set Samples:

```
data _null_;
   set samples;
   call symput('sample',strip(put(_n_,5.)));
   call symput('v1',strip(put(v1,5.)));
   call symput('v2',strip(put(v2,5.)));
   call symput('test',strip(test));
   call execute("%nrstr(%testsample)");
```

```
run;
%symdel sample v1 v2 test;
```

The DATA step uses the CALL SYMPUT routine to put variables SAMPLE, V1, V2, and TEST into global macro variables. It then uses the CALL EXECUTE routine to execute macro %TESTSAMPLE. The %NRSTR function delays the execution of macro %TESTSAMPLE until the end of a step, which is required for global macro variables SAMPLE, V1, V2, and TEST to resolve correctly. The %SYMDEL statement deletes the global macro variables when the DATA step is done.

When the DATA step is executed, the following is written to the SAS log:

```
Sample 1: V1=1, V2=17, Test is V1 <= V2: **TEST PASSED**
Sample 2: V1=7, V2=4, Test is V1 >= V2: **TEST PASSED**
Sample 3: V1=3, V2=1, Test is V1 = V2: **TEST FAILED**
Sample 4: V1=9, V2=5, Test is V1 >= V2: **TEST PASSED**
```

What to Do When Automatic Unquoting Does Not Work

When the macro processor generates text from an item masked by a macro quoting function, you can usually allow SAS to unquote the macro quoted items automatically. For example, suppose you define a macro variable PRINTIT:

```
%let printit=%str(proc print; run;);
```

Then you use that macro variable in your program:

```
%put *** This code prints the data set: &printit ***;
```

When the macro processor generates the text from the macro variable, the items masked by macro quoting functions are automatically unquoted, and the previously masked semicolons work normally when they are passed to the rest of SAS.

In rare cases, masking text with a macro quoting function changes how the word scanner tokenizes the text. (The word scanner and tokenization are discussed in Chapter 2, “SAS Programs and Macro Processing,” on page 17 and “Macro Processing” on page 45.) For example, a single or double quotation mark produced by resolution within the %BQUOTE function becomes a separate token. The word scanner does not use it as the boundary of a literal token in the input stack. If generated text that was once masked by the %BQUOTE function looks correct but SAS does not accept it, you might need to use the %UNQUOTE function to restore normal tokenization.

How Macro Quoting Works

When the macro processor masks a text string, it masks special characters and mnemonics within the coding scheme, and prefixes and suffixes the string with a hexadecimal character, called a *delta character*. The prefix character marks the beginning of the string and also indicates what type of macro quoting is to be applied to the string. The suffix character marks the end of the string. The prefix and suffix characters preserve any leading and trailing blanks contained by the string. The hexadecimal characters used to mask special characters and mnemonics and the characters used for the prefix and suffix might vary and are not portable.

There are more hexadecimal combinations possible in each byte than are needed to represent the symbols on a keyboard. Therefore, when a macro quoting function recognizes an item to be masked, the macro processor uses a previously unused hexadecimal combination for the prefix and suffix characters.

Macro functions, such as %EVAL and %SUBSTR, ignore the prefix and suffix characters. Therefore, the prefix and suffix characters do not affect comparisons.

When the macro processor is finished with a macro quoted text string, it removes the macro quoting-coded substitute characters and replaces them with the original characters. The unmasked characters are passed on to the rest of the system. Sometimes you might see a message about unmasking, as in the following example:

```
/* Turn on SYMBOLGEN so you can see the messages about unquoting. */
options symbolgen;

/* Assign a value to EXAMPLE that contains several special */
/* characters and a mnemonic. */
%let example = %nrbquote( 1 + 1 = 3 Today's Test and More );

%put *&example*;

/* Turn off SYMBOLGEN. */
options nosymbolgen;
```

When this program is submitted, the following appears in the SAS log:

```
SYMBOLGEN:   Macro variable EXAMPLE resolves to  1 + 1 = 3 Today's
             Test and More
SYMBOLGEN:   Some characters in the above value which were subject
             to macro quoting have been unquoted for printing.
* 1 + 1 = 3 Today's Test and More *
```

As you can see, the leading and trailing blanks and special characters were retained in the variable's value. When the macro processor was working with the string, the string actually contained coded characters that were substituted for the "real" characters. The substitute characters included coded characters to represent the start and end of the string. The leading and trailing blanks were preserved. Characters were also substituted for the special characters `+`, `=`, and `'`, and the mnemonic `AND`. When the macro finished processing and the characters were passed to the rest of SAS, the coding was removed and the real characters were replaced.

"Unquoting Text" on page 115 provides more information about what happens when a masked string is unquoted. For more information, see "SYMBOLGEN System Option" on page 477.

Other Functions That Perform Macro Quoting

Functions That Start with the Letter Q

Some macro functions are available in pairs, where one function starts with the letter Q:

- %SCAN and %QSCAN
- %SUBSTR and %QSUBSTR
- %UPCASE and %QUPCASE
- %SYSFUNC and %QSYSFUNC

The Q*xxx* functions are necessary because by default, macro functions return an unquoted result, even if the argument was masked by a macro quoting function. The %QSCAN, %QSUBSTR, %QUPCASE, and %QSYSFUNC functions mask the returned value at execution time. The items masked are the same as the items masked by the %NRBQUOTE function.

Example Using the %QSCAN Function

This example uses the %QSCAN function to parse a delimited list of shop names in macro variable SHOPS into separate macro variables, one for each shop name. The percent sign (%) is used as the delimiter. For this example, here is a simple DATA step that creates macro variable SHOPS:

```
data _null_;
   input shops $1-80;
   call symput("shops", shops);
datalines;
Fischer Books%Smith&Sons%Sarah's Sweet Shoppe
;
run;
```

The following macro uses the %QSCAN function to assign shop names in the value of macro variable SHOPS to separate macro variables, one for each name.

```
%macro splitit(val);
   %local i;
   %do i=1 %to 3;
      %global x&i;
      %let x&i=%qscan(&val,&i,%);
   %end;
%mend splitit;
```

The iterative %DO loop creates a global macro variable for each item in SHOPS and assigns it the value of that item.

Here is an example of the %SPLITIT macro execution.

```
%splitit(%superq(shops));
```

The %SUPERQ function masks the value of SHOPS in the %SPLITIT macro call, which prevents any resolution of the value of SHOPS. The three values are stored in global macro variables X1, X2, and X3 as shown in the following.

```
%put &x1; %put &x2; %put &x3;
```

```
Fischer Books
Smith&Sons
Sarah's Sweet Shoppe
```

8

Interfaces with the Macro Facility

Interfaces with the Macro Facility

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. The connection between the macro facility and the rest of SAS is not usually dynamic. But by using an interface to the macro facility, you can dynamically connect the macro facility to the rest of SAS.

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. These elements of the macro language are not considered true macro facility interfaces and they are discussed in this section. For more information about these macro language elements, see Chapter 12, "Macro Language Elements," on page 189.

DATA Step Interfaces

Interacting with the Macro Facility during DATA Step Execution

DATA step interfaces consist of eight 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. You can use one of the DATA step interfaces to interact with the macro facility during DATA step execution. You can use DATA step interfaces to do the following:

- 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
- delete a macro variable
- pass information about a macro variable from the macro facility to the DATA step

The following table lists the DATA step interfaces by category and their uses.

Table 8.1 DATA Step Interfaces to the Macro Facility

Category	Tool	Description
Execution	CALL EXECUTE routine	Resolves its argument and executes the resolved value at the

Category	Tool	Description
		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.
Deletion	CALL SYMDEL routine	Deletes the indicated macro variable named in the argument.
Information	SYMEXIST function	Returns an indication as to whether the macro variable exists.
Read or Write	SYMGET function	Returns the value of a macro variable during DATA step execution.
Information	SYMGLOBL function	Returns an indication as to whether the macro variable is global in scope.
Information	SYMLOCAL function	Returns an indication as to whether the macro variable is local in scope.
Read or Write	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: 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. See the examples that follow and SAS Usage Note 23134 for more information.

Example of Using CALL EXECUTE Incorrectly

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

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

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

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

The %GLOBAL statement is not necessary in this version. Because the %LET statement is executed in open code, it automatically creates a global macro variable. (For more information about macro variable scopes, see Chapter 5, “Scopes of Macro Variables,” on page 57.)

Example of a Common Problem with 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 $;
   datalines;
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 during 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, which is 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;
   datalines;
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.

Example of a Problem with CALL EXECUTE and Macro Variable References in Macros

This example shows a problem that can occur when CALL EXECUTE invokes a macro that contains references to macro variables that were created using CALL SYMPUT in that macro. Unlike the previous examples, the problem in this example occurs in the macro itself and not in the DATA step in which it is invoked.

```
data prices;
   length product $30;
   input code amount discount product $;
   cards;
```

```
56 300   0.2  Skates
99 10000 .    Boats
24 225   0.25 Footballls
;
run;

%macro onSale(product=,price=,discount=);
   data _null_;
      regularPrice = &price;
      salePrice = round(&price - (&price*&discount));
      call symputx('salePrice', put(salePrice,dollar12.));
      call symputx('regularPrice', put(regularPrice,dollar12.));
   run;
   %put NOTE: &product marked down from &regularPrice to &salePrice..;
%mend onSale;

/* macro variables SALEPRICE and REGULARPRICE fail to resolve in
%onSale */
data sales;
   set prices;
   if (discount) then do;
      call execute('%onSale(product='||product||',price='||
vvalue(amount)||
         ',discount='||vvalue(discount)||')');
   end;
run;
```

Macro %ONSALE uses CALL SYMPUT statements in a _NULL_ DATA step to create macro variables SALEPRICE and REGULARPRICE. It includes references to both macro variables in the %PUT statement that follows the DATA step. These macro variables are not used in the SALES DATA step, so the issue described in the previous examples does not apply, in this case. In the SALES DATA step in this example, CALL EXECUTE is used to invoke %ONSALE for each product that is discounted. As demonstrated in the previous examples, macro %ONSALE executes during SALES DATA step compilation. As a result, the references to macro variables SALEPRICE and REGULARPRICE in %ONSALE fail to resolve because they are not available.

To solve this issue, use %NRSTR to mask the %ONSALE macro call in the CALL EXECUTE argument during the SALES DATA step compilation. Here is the SALES DATA step modified to use %NRSTR:

```
/* macro variables SALEPRICE and REGULARPRICE resolve in %onSale */
data sales;
   set prices;
   if (discount) then do;
      call execute('%nrstr(%onSale(product='||product||',price='||
         vvalue(amount)||',discount='||vvalue(discount)||'))');
   end;
run;
```

Using %NRSTR delays the execution of macro %ONSALE until after the SALES DATA step executes. In that case, references to macro variables SALEPRICE and REGULARPRICE in %ONSALE resolve correctly. For more information about using %NRSTR in this case, see SAS Usage Note 23134.

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 about these functions, see "%SYSFUNC Macro Function" on page 356.

%SYSFUNC arguments are a single SAS language function and an optional format. See the following examples:

```
%put %sysfunc(date(),worddate.);
%put %sysfunc(exist(work.mydata));
```

All arguments in SAS language functions within %SYSFUNC must be separated by commas. You cannot use argument lists preceded by the word OF. The arguments to %SYSFUNC are evaluated according to the rules of the SAS macro language. This includes both the function name and the argument list to the function. When a function that is called by %SYSFUNC requires a numeric argument, the macro facility converts the argument to a numeric value. %SYSFUNC can return a floating-point number when returned by the function that it executes. See "%SYSFUNC Macro Function" on page 356 for more information.

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

```
%put %sysfunc(intnx(month,%sysfunc(date()),0,end));
```

This example returns the date value of the current date incremented by one month and aligned to the end of the month. The nested %SYSFUNC function invokes DATE() and returns the current date string to the INTNX function. The INTNX function then increments the date by one month, aligns it to the end of the month, and returns the new date value. Notice that function INTNX parameter values MONTH and END are not enclosed in quotation marks, even though they are character values.

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));`

There are situations where you might need to mask special characters in argument values, as shown in the following example:

```
%put %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 (%). In some cases, you might need to specify a blank space as an argument value, as shown in the following example:

```
%let colors=red green blue orange yellow;
%put %sysfunc(countw(&colors,%str( )));
```

In this example, function COUNTW returns the number of words in macro variable COLORS using a blank space as the list delimiter. The %STR function masks the blank space.

You can use %SYSFUNC and %QSYSFUNC to call all of the DATA step SAS functions except the ones that are listed in Table 17.49 on page 332. 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 “Macro Statements” on page 378.

Interfaces with the SQL Procedure

Using PROC SQL

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

- 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. For a summary of how macro variables are created, see . on page 27 For more information and examples relating to the INTO clause, see Chapter 3, “Macro Variables,” on page 27.

Controlling Job Execution

PROC SQL also provides macro tools to do the following:

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

The following table provides information about macro variables created by SQL that affect job execution.

Table 8.2 Macro Variables That Affect Job Execution

Macro Variable	Description
SQLEXITCODE	Contains the highest return code that occurred from some types of SQL insert failures. This return code is written to the SYSERR macro variable when PROC SQL terminates.
SQLOBS	Contains the number of rows or observations produced by a SELECT statement.
SQLOOPS	Contains the number of iterations that the inner loop of PROC SQL processes.
SQLRC	Contains the return code from an SQL statement. For return codes, see SAS SQL documentation.
SQLXMSG	Contains descriptive information and the DBMS-specific return code for the error that is returned by the pass-through facility.
SQLXRC	contains the DBMS-specific return code that is returned by the pass-through facility.

Interfaces with the SAS Component Language

Using an SCL Program

You can use the SAS macro facility to define macros and macro variables for an SCL program. Then, you can pass parameters between macros and the rest of the 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 might 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.

The following table 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 variable.
	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 more information about these elements, see "DATA Step Call Routines for Macros" on page 283 and "DATA Step Functions for Macros" on page 299.

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 SAS 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 that 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.

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

Overview of SAS/CONNECT Interfaces

Many times when working with macros and SAS/CONNECT, the results that you see are not what you expected. When using RSUBMIT within the macro facility, it is important to have an understanding of what happens at compile time versus what happens at execution time. Knowing the behavior of this interaction helps you when using macros and SAS/CONNECT together.

Using %SYSRPUT with SAS/CONNECT

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 Compute Server 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 windowing environment session by submitting the DMR option with the SAS command. For more information about using %SYSRPUT, see "The %SYSLPUT and %SYSRPUT Statements" in *SAS/CONNECT User's Guide*.

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 information and syntax of the %SYSRPUT statement, see "%SYSRPUT Macro Statement" on page 429.

Using %SYSLPUT with SAS/CONNECT

The %SYSLPUT statement is a macro statement that is submitted in the client session to assign a value that is available in the client session to a macro variable that can be accessed from the server session. If you are signed on to multiple server sessions, %SYSLPUT submits the macro assignment statement to the most recently used server session. If you are signed on to only one server session, %SYSLPUT submits the macro assignment statement to that server session. If you are not signed on to any session, an error condition results. Like the %LET statement, the %SYSLPUT statement assigns a value to a macro variable. Unlike %LET, the %SYSRPUT statement assigns a value to a variable in the server session rather than in the client session where the statement is executed. The %SYSRPUT statement stores the macro variable in the Global Symbol Table in the server session.

For more information about using %SYSLPUT, see "The %SYSLPUT and %SYSRPUT Statements" in *SAS/CONNECT User's Guide*.

Example Using %SYSLPUT

%SYSLPUT enables you to dynamically assign values to variables that are used by macros that are executed in a server session. The macro statement %SYSLPUT is used to create the macro variable REMID in the server session and to use the value of the client macro variable RUNID. The REMID variable is used by the %DOLIB macro, which is executed in a server session. This process finds out which operating system-specific library assignment should be used in the server session.

```
%macro assignlib (runid);

   signon rem &runid
   %syslput remid=&runid
   rsubmit rem &runid
      %macro dolib;
         %if (&runid eq 1) %then %do;
            libname mylib 'h:';
            %end;
         %else %if (&runid eq 2) %then %do;
            libname mylib '/afs/some/unix/path';
            %end;
      %mend;
      %dolib;
   endrsubmit;

%mend;
```

9

Storing and Reusing Macros

Storing and Reusing Macros

When you submit a macro definition, by default, the macro processor compiles and stores the macro in a SAS catalog in the Work library. These macros, referred to as *session compiled macros*, exist only during the current Compute Server session. To save frequently used macros between sessions, you can use either the autocall macro facility or the stored compiled macro facility.

The autocall macro facility stores the source for SAS macros in a collection of external files called an *autocall library*. The autocall facility is useful when you want to create a pool of easily maintained macros in a location that can be accessed by different applications and users. Autocall libraries can be concatenated together. The primary disadvantage of the autocall facility is that the first time an autocall macro is called in a session, the macro processor compiles it. This compilation is overhead that you can avoid by using the stored compiled macro facility.

The stored compiled macro facility stores compiled macros in a SAS catalog in a SAS library that you specify. By using stored compiled macros, you might save macro compilation time in your production-level jobs. However, because these stored macros are compiled, you must save and maintain the source for the macro definitions in a different location.

The autocall facility and the stored compiled macro facility each offer advantages. Here are some of the factors that determine how you choose to save a macro definition:

- how often you use a macro
- how often you change it
- how many users need to execute it
- how many compiled macro statements it has

If you are developing new programs, consider creating macros and compiling them during your current session. If you are running production-level jobs using name-style macros, consider using stored compiled macros. If you are letting a group of users share macros, consider using the autocall facility.

Note: For greater efficiency, store only name-style macros if you use the stored compiled macro facility. Storing statement-style and command-style macros is less efficient.

It is good practice, when you are programming stored compiled macros or autocall macros, to use the %LOCAL statement. This statement defines macro variables that will be used only inside that macro. Otherwise, values of macro variables defined outside of the current macro might be altered. See the discussion of macro variable scopes in Chapter 5, “Scopes of Macro Variables,” on page 57.

In general, macro and variable names in the SAS macro facility are case insensitive and are internally changed to uppercase. The values are case sensitive in the SAS macro facility and are not changed.

When calling an autocall macro or a stored compiled macro, the macro name is changed to uppercase and passed to the catalog routines to open a member of that name. The catalog routines are host dependent and use the default casing for the particular host when searching for a member. Macro catalog entries should be made using the default casing for the host in question. Linux default is lowercase.

Note: In Linux, the member name that contains the autocall macro must be all lowercase letters.

Saving Macros in an Autocall Library

Overview of an Autocall Library

Generally, an autocall library is a directory containing individual files, each of which contains one macro definition. In SAS 6.11 and later, an autocall library can also be a SAS catalog. (See the following section for more information about using SAS catalogs as autocall libraries.)

Operating Environment Information: *Autocall Libraries on Different Hosts* The term *directory* refers to an aggregate storage location that contains files (or members) managed by the host operating system. Different host operating systems identify an aggregate storage location with different names, such as a directory, a subdirectory, a maclib, a text library, or a partitioned data set. For more information, see the SAS Companion for your operating system.

Using Directories as Autocall Libraries

To use a directory as a SAS autocall library, do the following:

1. To create library members, store the source code for each macro in a separate file in a directory. The name of the file must be the same as the macro name. For example, the statements defining a macro that you would call by submitting %SPLIT must be in a file named Split.

   Operating Environment Information: *Autocall Library Member Names* On operating systems that allow filenames with extensions, you must name autocall macro library members with a special extension, usually `.SAS`. Look at the autocall macros on your system provided by SAS to determine whether names of files containing macros must have a special extension at your site. On z/OS operating systems, you must assign the macro name as the name of the PDS member.

2. Set the SASAUTOS system option to specify the directory as an autocall library. On most hosts, the reserved fileref SASAUTOS is assigned at invocation time to the autocall library supplied by SAS or another one designated by your site. If you are specifying one or more autocall libraries, remember to concatenate the autocall library supplied by SAS with your autocall libraries so that these macros will also be available. For more information, see your host documentation and “SASAUTOS= System Option” on page 472.

When storing files in an autocall library, remember the following:

- SAS does not restrict the type of material that you place in an autocall library. You should store only autocall library files in it to avoid confusion and for ease of maintenance.
- SAS lets you include more than one macro definition, as well as open code, in an autocall library member. You should generally keep only one macro in any autocall library member. If you need to keep several macros in the same autocall library member, keep related macros together.

Using SAS Catalogs as Autocall Libraries

In SAS 6.11 and later, you can use the CATALOG access method to store autocall macros as SOURCE entries in SAS catalogs. To create an autocall library using a SAS catalog, follow these steps:

1. Use a LIBNAME statement to assign a libref to the SAS library.
2. Use a FILENAME statement with the CATALOG argument to assign a fileref to the catalog that contains the autocall macros. For example, the following code creates a fileref, MyMacros, that points to a catalog named MyMacs.MyAutos:

   ```
   libname mymacs 'SAS-library';
   filename mymacros catalog 'mymacs.myautos';
   ```

3. Store the source code for each macro in a SOURCE entry in a SAS catalog. (SOURCE is the entry type.) The name of the SOURCE entry must be the same as the macro name.
4. Set the SASAUTOS system option to specify the fileref as an autocall library. For more information, see "SASAUTOS= System Option" on page 472.

Calling an Autocall Macro

To call an autocall macro, the system options MAUTOSOURCE must be set and SASAUTOS must be assigned. MAUTOSOURCE enables the autocall facility, and SASAUTOS specifies the autocall libraries. For more information, see "MAUTOSOURCE System Option" on page 442 and "SASAUTOS= System Option" on page 472.

Once you have set the required options, calling an autocall macro is like calling a macro that you have created in your current session. However, it is important that you understand how the macro processor locates the called macro. When you call a macro, the macro processor does the following tasks:

- searches for a session compiled macro definition
- searches for a stored compiled macro definition in the library specified by the SASMSTORE option, if the MSTORED option is set

- searches for a member in the autocall libraries specified by the SASAUTOS option in the order in which they are specified, if the MAUTOSOURCE option is set
- searches the SASHelp library for SAS production stored compiled macro definitions

When SAS finds a library member in an autocall library with that macro name, the macro processor does the following:

- compiles all of the source statements in that member, including any and all macro definitions, and stores the result in the session catalog
- executes any open code (macro statements or SAS source statements not within any macro definition) in that member
- executes the macro with the name that you invoked

Note: If an autocall library member contains more than one macro, the macro processor compiles all of the macros but executes only the macro with the name that you invoked.

Any open code statements in the same autocall library member as a macro execute only the first time you invoke the macro. When you invoke the macro later in the same session, the compiled macro is executed, which contains only the compiled macro definition and not the other code the autocall macro source file might have contained.

It is not advisable to change SASAUTOS during a Compute Server session. If you change the SASAUTOS= specification in an ongoing session, SAS will store the new specification only until you invoke an uncompiled autocall macro. SAS then will close all opened libraries and open all the newly specified libraries that it can open.

For information about debugging autocall macros, see Chapter 10, “Macro Facility Error Messages and Debugging,” on page 145.

Saving Macros Using the Stored Compiled Macro Facility

Overview of the Stored Compiled Macro Facility

The stored compiled macro facility compiles and saves compiled macros in a permanent catalog in a library that you specify. This compilation occurs only once. If the stored compiled macro is called in the current or later sessions, the macro processor executes the compiled code.

The stored compiled macro catalog is initially opened for Read-Only access. When a stored compiled macro is being compiled or updated, the catalog is immediately closed and reopened for Update access. After the macro is compiled and the catalog has been updated or changed, the catalog is again immediately closed and reopened for Read-Only access.

Compiling and Storing a Macro Definition

To compile a macro definition in a permanent catalog, you must first create the source for each stored compiled macro. To store the compiled macro, use the following steps:

1 Use the STORE option in the %MACRO statement. You can use the SOURCE option to store the source code with the compiled code. In addition, you can assign a descriptive title for the macro entry in the SAS catalog, by specifying the DES= option. For example, the %MACRO statement in the following definition shows the STORE, SOURCE, and DES= options:

```
%macro myfiles / store source
     des='Define filenames';
  filename file1 'external-file-1';
  filename file2 'external-file-2';
%mend;
```

CAUTION
Save your macro source code. You cannot re-create the source statements from a compiled macro. Therefore, you must save the original macro source statements if you want to change the macro. For all stored compiled macros, you should document your macro source code well. You can save the source code with the compiled code using the SOURCE option in the %MACRO statement or you can save the source in a separate file. If you save the source in a separate file, it is recommended that you save the source code in the same catalog as the compiled macro. In this example, save it to the following library:

```
mylib.sasmacro.myfiles.source
```

Note: To retrieve the source of a compiled stored macro, see "%COPY Macro Statement" on page 385.

2 Set the MSTORED system option to enable the stored compiled macro facility. For more information, see "MSTORED System Option" on page 469.

3 Assign the SASMSTORE option to specify the SAS library that contains or will contain the catalog of stored compiled SAS macros. For example, to store or call compiled macros in a SAS catalog named MyLib.SASMacr, submit these statements:

```
libname mylib 'SAS-library';
options mstored sasmstore=mylib;
```

For more information, see "SASMSTORE= System Option" on page 475.

4 Submit the source for each macro that you want to compile and permanently store.

You should not move a stored compiled macro to another operating system or to a different release of SAS. This scenario is not supported. If you execute a stored compiled macro that was compiled on a different operating system or with a different release of SAS, a warning message is written to the SAS log. The warning causes a SAS job to return a status code of 1 instead of 0. You might also encounter other unexpected issues with your stored compiled macros. To avoid issues, you can move or copy the macro source code to your current SAS system, and then compile and store it on your current SAS system.For more information, see your host companion.

TIP You can set system option NOMACROCOMPVERWARN to write a note to the SAS log instead of a warning message in this case. However, this should be used as a temporary measure until the stored compiled macros can be recompiled in the current version of SAS, if possible. See "MACROCOMPVERWARN System Option" on page 437.

Storing Autocall Macros Supplied by SAS

If you use the macros in the autocall library supplied by SAS, you can save macro compile time by compiling and storing those macros in addition to ones that you create yourself. Many of the macros related to Base SAS software that are in the autocall library supplied by SAS can be compiled and stored in a SAS catalog named SASMacr. This action can be done by using the autocall macro COMPSTOR that is supplied by SAS. For more information, see "%COMPSTOR Autocall Macro" on page 215.

Calling a Stored Compiled Macro

Once you have set the required system options, calling a stored compiled macro is just like calling session compiled macros. However, it is important that you understand how the macro processor locates a macro. When you call a macro, the macro processor searches for the macro name using this sequence:

1 the macros compiled during the current session

2 the stored compiled macros in the SASMacr catalog in the specified library (if options MSTORED and SASMSTORE= are in effect)

3 each autocall library specified in the SASAUTOS option (if options SASAUTOS= and MAUTOSOURCE are in effect)

4 the stored compiled macros in the SASMacr catalog in the SASHelp library

You can display the entries in a catalog containing compiled macros. For more information, see Chapter 10, “Macro Facility Error Messages and Debugging,” on page 145.

10

Macro Facility Error Messages and Debugging

General Macro Debugging Information

Developing Macros in a Layered Approach

Because the macro facility is such a powerful tool, it is also complex, and debugging large macro applications can be extremely time-consuming and frustrating. Therefore, it makes sense to develop your macro application in a way that minimizes the errors. This makes the errors that do occur as easy as possible to find and fix. The first step is to understand what type of errors can occur and when they manifest themselves. Then, develop your macros using a modular, layered approach. Finally, use some built-in tools such as system options, automatic macro variables, and the %PUT statement to diagnose errors.

Note: To receive certain important warning messages about unresolved macro names and macro variables, be sure the "SERROR System Option" on page 476 and "MERROR System Option" on page 449 are in effect. For more information, see "System Options for Macros" on page 434.

Encountering Errors

When the word scanner processes a program and finds a token in the form of `&` or `%`, it triggers the macro processor to examine the name token that follows the `&` or `%`. Depending on the token, the macro processor initiates one of the following activities:

- macro variable resolution
- macro open code processing
- macro compilation
- macro execution

An error can occur during any one of these stages. For example, if you misspell a macro function name or omit a necessary semicolon, that is a *syntax error* during compilation. Syntax errors occur when program statements do not conform to the rules of the macro language. Or, you might refer to a variable out of scope, causing a *macro variable resolution error*. *Execution errors* (also called *semantic errors*) are usually errors in program logic. They can occur, for example, when the text generated by the macro has faulty logic (statements not executed in the right order or in the way you expect).

Of course, even if your macro code is perfect, that does not guarantee that you will not encounter errors caused by plain SAS code. For example, you might encounter the following:

- a libref that is not defined
- a syntax error in open code (that is, outside of a macro definition)
- a typographical error in the code that your macro generates

Typically, error messages with numbers are plain SAS code error messages. Error messages generated by the macro processor do not have numbers.

Developing Bug-free Macros

When programming in any language, it is good technique to develop your code in modules. That is, instead of writing one massive program, develop it piece by piece, test each piece separately, and put the pieces together. This technique is especially useful when developing macro applications because of the two-part nature of SAS macros: macro code and the SAS code generated by the macro code.

Another good idea is to proofread your macro code for common mistakes before you submit it.

The following list outlines some key items to check for:

- The names in the %MACRO and %MEND statements match, and there is a %MEND for each %MACRO.
- The number of %DO statements matches the number of %END statements.
- %TO values for iterative %DO statements exist and are appropriate.
- All statements end with semicolons.
- Comments begin and end correctly and do not contain unmatched single quotation marks.
- Macro variable references begin with `&`and macro statements begin with `%`.
- Macro variables created by CALL SYMPUT are not referenced in the same DATA step in which they are created.
- Statements that execute immediately (such as %LET) are not part of conditional DATA step logic.
- Single quotation marks are not used around macro variable references (such as in TITLE or FILENAME statements). When used in quoted strings, macro variable references resolve only in strings marked with double quotation marks.
- Macro variable values do not contain any keywords or characters that could be interpreted as mathematical operators. (If they do contain such characters, use the appropriate macro quoting function.)
- Macro variables, %GOTO labels, and macro names do not conflict with reserved SAS and host environment keywords.

Troubleshooting Your Macros

Solving Common Macro Problems

The following table lists some problems that you might encounter when working with the macro facility. Because many of these problems do not cause error messages to be written to the SAS log, solving them can be difficult. For each problem, the table gives some possible causes and solutions.

Table 10.1 Commonly Encountered Macro Problems

Problem	Cause	Explanation
SAS windowing environment session stops responding after you submit a macro definition. You enter and submit code but nothing happens.	■ Syntax error in %MEND statement ■ Missing semicolon, parenthesis, or quotation mark ■ Missing %MEND statement ■ Unclosed comment	The %MEND statement is not recognized and all text is becoming part of the macro definition.
SAS windowing environment session stops responding after you call a macro.	An error in invocation, such as forgetting to provide one or more parameters, or forgetting to use parentheses when invoking a macro that is defined with parameters.	The macro facility is waiting for you to finish the invocation.
The macro does not compile when you submit it.	A syntax error exists somewhere in the macro definition.	Only syntactically correct macros are compiled.
The macro does not execute when you call it or partially executes and stops.	■ A bad value was passed to the macro (for example, as a parameter). ■ A syntax error exists somewhere in the macro definition.	A macro successfully executes only when it receives the correct number of parameters that are of the correct type.

Problem	Cause	Explanation
The macro executes but the SAS code gives bad results or no results.	Incorrect logic in the macro or SAS code.	
Code runs fine if submitted as open code, but when generated by a macro, the code does not work and issues strange error messages.	■ Tokenization is not as you intended. ■ A syntax error exists somewhere in the macro definition.	Rarely, macro quoting functions alter the tokenization of text enclosed in them. Use the "%UNQUOTE Macro Function" on page 370.
A %MACRO statement generates "invalid statement" error.	■ The MACRO system option is turned off. ■ A syntax error exists somewhere in the macro definition.	For the macro facility to work, the MACRO system option must be on. Edit your SAS configuration file accordingly.

The following table lists some common macro error and warning messages. For each message, some probable causes are listed, and pointers to more information are provided.

Table 10.2 Common Macro Error Messages and Causes

Error Message	Possible Causes	For More Information
`Apparent invocation of macro xxx not resolved.`	■ You have misspelled the macro name. ■ MAUTOSOURCE system option is turned off. ■ MAUTOSOURCE is on, but you have specified an incorrect pathname in the SASAUTOS= system option. ■ You are using the autocall facility but have given the macro and file different names. ■ You are using the autocall facility but did not give the file the **`.sas`** extension.	■ Check the spelling of the macro name. ■ "Solving Problems with the Autocall Facility" on page 158. ■ "Developing Bug-free Macros" on page 147.

Error Message	Possible Causes	For More Information
	■ There is a syntax error within the macro definition.	
`Apparent symbolic reference xxx not resolved.`	■ You are trying to resolve a macro variable in the same DATA step as the CALL SYMPUT that created it. ■ You have misspelled the macro variable name. ■ You are referencing a macro variable that is not in scope. ■ You have omitted the period delimiter when adding text to the end of the macro variable.	■ “Resolving Timing Issues” on page 155. ■ Check the spelling of the macro variable. ■ “Solving Problems with Macro Variable Scope” on page 151. ■ “Solving Macro Variable Resolution Problems” on page 150. ■ “Generating a Suffix for a Macro Variable Reference” on page 14

For a list of macro facility error messages and warnings, see “SAS Macro Error Messages” on page 485 and “SAS Macro Warning Messages” on page 521.

Solving Macro Variable Resolution Problems

When the macro processor examines a name token that follows an `&`, it searches the macro symbol tables for a matching macro variable entry. If it finds a matching entry, it pulls the associated text from the symbol table and replaces `&name` on the input stack. A macro variable name is passed to the macro processor, but the processor does not find a matching entry in the symbol tables. So, it leaves the token on the input stack and generates this message:

```
WARNING: Apparent symbolic reference
NAME not resolved.
```

The unresolved token is transferred to the input stack for use by other parts of SAS.

Note: You receive the WARNING only if the SERROR system option is on.

To solve these problems, check that you have spelled the macro variable name right and that you are referencing it in an appropriate scope.

When a macro variable resolves but does not resolve to the correct value, you can check several things. First, if the variable is a result of a calculation, ensure that the correct values were passed into the calculation. And, ensure that you have not inadvertently changed the value of a global variable. (For more information about variable scope problems, see "Solving Problems with Macro Variable Scope" on page 151.)

Another common problem is adding text to the end of a macro variable but forgetting to add a delimiter that shows where the macro variable name ends and the added text begins. For example, suppose you want to write a TITLE statement with a reference to WEEK1, WEEK2, and so on. You set a macro variable equal to the first part of the string and supply the week's number in the TITLE statement:

```
%let wk=week;
%put This is data for &wk1.;   /* INCORRECT */
```

When these statements compile, the macro processor looks for a macro variable named WK1, not WK. To fix the problem, add a period (the macro delimiter) between the end of the macro variable name and the added text, as in the following statements:

```
%let wk=week;
%put This is data for &wk.1.;
```

CAUTION

Do not use AF, DMS, or SYS as prefixes with macro variable names. The letters AF, DMS, and SYS are frequently used by SAS as prefixes for macro variables created by SAS. SAS does not prevent you from using AF, DMS, or SYS as a prefix for macro variable names. However, using these strings as prefixes might create a conflict between the names that you specify and the name of a SAS created macro variable (including automatic macro variables in later SAS releases). If a name conflict occurs, SAS might not issue a warning or error message, depending on the details of the conflict. Therefore, the best practice is to avoid using the strings AF, DMS, or SYS as the beginning characters of macro names and macro variable names.

Solving Problems with Macro Variable Scope

A common mistake that occurs with macro variables concerns referencing local macro variables outside of their scope. As described in Appendix 1, " Reserved Words in the Macro Facility," on page 483 macro variables are either global or local. Referencing a variable outside of its scope prevents the macro processor from resolving the variable reference. For example, suppose that you have the following data set:

Example Code 10.1 *Sasuser.Houses DATA Step*

```
data sasuser.houses;
input style $ 1-9 sqfeet 10-13 bedrooms 15 baths 17-19 street $ 21-36
price 38-44;
format price dollar8.;
datalines;
RANCH    1250 2 1.0 Sheppart Avenue   64000
SPLIT    1190 1 1.0 Rand Street       65850
CONDO    1400 2 1.5 Market Street     80050
```

```
TWOSTORY 1810 4 3.0 Garris Street    107250
RANCH    1500 3 3.0 Kemble Avenue     86650
SPLIT    1615 4 3.0 West Drive        94450
SPLIT    1305 3 1.5 Graham Avenue     73650
CONDO    1390 3 2.5 Hampshire Avenue  79650
TWOSTORY 1040 2 1.0 Sanders Road      55850
CONDO    2105 4 2.5 Jeans Avenue     127150
RANCH    1535 3 3.0 State Highway     89100
TWOSTORY 1240 2 1.0 Fairbanks Circle  69250
RANCH     720 1 1.0 Nicholson Drive   34550
TWOSTORY 1745 4 2.5 Highland Road    102950
CONDO    1860 2 2.0 Arcata Avenue    110700
run;
```

Now, consider the following program:

```
%macro totinv(var);
   data inv;
      retain total 0;
      set Sasuser.Houses end=final;
      total=total+&var;
      if final then call symput("macvar",put(total,dollar14.2));
   run;
   %put **** TOTAL=&macvar ****;
%mend totinv;

%totinv(price)
%put **** TOTAL=&macvar ****;   /* ERROR */
```

When you submit these statements, the DATA step creates data set Sasuser.Houses. The %PUT statement in the macro TOTINV writes the value of TOTAL to the log. The %PUT statement that follows the macro call generates a warning message and writes the text `TOTAL=&macvar` to the log, as follows:

```
TOTAL= $1,241,100.00
WARNING: Apparent symbolic reference MACVAR not resolved.
**** TOTAL=&macvar ****
```

The second %PUT statement fails because the macro variable MACVAR is local to the TOTINV macro. To correct the error, you must use a %GLOBAL statement to declare the macro variable MACVAR.

Another common mistake that occurs with macro variables concerns overlapping macro variable names. If, within a macro definition, you refer to a macro variable with the same name as a global macro variable, you affect the global variable, which might not be what you intended. Either give your macro variables distinct names or use a %LOCAL statement to specifically define the variables in a local scope. For an example of this technique, see “Forcing a Macro Variable to Be Local” on page 72.

Solving Open Code Statement Recursion Problems

Recursion is something calling itself. *Open code recursion* is when your open code erroneously causes a macro statement to call another macro statement. This call is referred to as a recursive reference. The most common error that causes open code recursion is a missing semicolon. In the following example, the %LET statement is not terminated by a semicolon:

```
   %let a=b   /* ERROR */
   %put **** &a ****;
```

When the macro processor encounters the %PUT statement within the %LET statement, it generates this error message:

```
   ERROR: Open code statement recursion detected.
```

Open code recursion errors usually occur because the macro processor is not reading your macro statements as you intended. Careful proofreading can usually solve open code recursion errors, because this type of error is mostly the result of typographical errors in your code, not errors in execution logic.

To recover from an open code recursion error, first try submitting a single semicolon. If that does not work, try submitting the following string:

```
*'; *"; *); */; %mend; run;
```

Continue submitting this string until the following message appears in the SAS log:

```
   ERROR: No matching %MACRO statement for this %MEND statement.
```

If the above method does not work, close your SAS session, and then restart it. Of course, closing and restarting your SAS session causes you to lose any unsaved data. Be sure to save often while you are developing your macros, and proofread them carefully before you submit them.

Solving Problems with Macro Functions

Some common causes of problems with macro functions include the following:

- misspelling the function name
- omitting the opening or closing parenthesis
- omitting an argument or specifying an extra argument

If you encounter an error related to a macro function, you might also see other error messages. The messages are generated by the invalid tokens left on the input stack by the macro processor.

Consider the following example. The user wants to use the %SUBSTR function to assign a portion of the value of the macro variable LINCOLN to the macro variable SECONDWD. But a typographical error exists in the second %LET statement, where %SUBSTR is misspelled as %SUBSRT:

```
%macro test;
   %let lincoln=Four score and seven;
   %let secondwd=%subsrt(&lincoln,6,5);   /* ERROR */
   %put *** &secondwd ***;
%mend test;

%test
```

When the erroneous program is submitted, the following appears in the SAS log:

```
WARNING: Apparent invocation of macro SUBSRT not resolved.
```

The error messages clearly point to the function name, which is misspelled.

Solving Unresolved Macro Problems

When a macro name is passed to the macro processor but the processor does not find a matching macro definition, it generates the following message:

```
WARNING: Apparent invocation of macro
NAME not resolved.
```

This error could be caused by the following:

- the misspelling of the name of a macro or a macro function
- an error in a macro definition that caused the macro to be compiled as a dummy macro

A *dummy macro* is a macro that the macro processor partially compiles but does not store.

Note: You receive this warning only if the MERROR system option is on.

Solving the “Black Hole” Macro Problem

When the macro processor begins compiling a macro definition, it reads and compiles tokens until it finds a matching %MEND statement. If you omit a %MEND statement or cause it to be unrecognized by omitting a semicolon in the preceding statement, the macro processor does not stop compiling tokens. Every line of code that you submit becomes part of the macro.

Resubmitting the macro definition and adding the %MEND statement does not correct the error. When you submit the corrected definition, the macro processor treats it as a nested definition in the original macro definition. The macro processor must find a matching %MEND statement to stop compilation.

Note: It is a good practice to use the %MEND statement with the macro name, so you can easily match %MACRO and %MEND statements.

If you recognize that SAS is not processing submitted statements and you are not sure how to recover, submit %MEND statements one at a time until the following message appears in the SAS log:

```
ERROR: No matching %MACRO statement for this %MEND statement.
```

Then recall the original erroneous macro definition, correct the error in the %MEND statement, and submit the definition for compilation.

There are other syntax errors that can create similar problems, such as unmatched quotation marks and unclosed parentheses. Often, one of these syntax errors leads to others. Consider the following example:

```
%macro rooms;
   /* other macro statements& */
   %put **** %str(John's office) ****;   /* ERROR */
%mend rooms;

%rooms
```

When you submit these statements, the macro processor begins to compile the macro definition ROOMS. However, the single quotation mark in the %PUT statement is not marked by a percent sign. Therefore, during compilation the macro processor interprets the single quotation mark as the beginning of a literal token. It does not recognize the closing parenthesis, the semicolon at the end of the statement, or the %MEND statement at the end of the macro definition.

To recover from this error, you must submit the following:

```
');
%mend;
```

If the above methods do not work, try submitting the following string:

```
*'; *"; *); */; %mend; run;
```

Continue submitting this string until the following message appears in the SAS log:

```
ERROR: No matching %MACRO statement for this %MEND statement.
```

Obviously, it is easier to catch these errors before they occur. You can avoid subtle syntax errors by carefully checking your macros before submitting them for compilation. For a syntax checklist, see "Developing Bug-free Macros" on page 147.

Note: Another cause of unexplained and unexpected macro behavior is using a reserved word as the name of a macro variable or macro. For example, because SAS reserves names starting with SYS, you should not create macros and macro variables with names beginning with SYS. Most host environments have reserved words too. For example, the word CON is reserved for console input. For reserved SAS keywords, see Appendix 1, " Reserved Words in the Macro Facility," on page 483. Check your SAS companion for host environment reserved words.

Resolving Timing Issues

Many macro errors occur because a macro variable resolves at a different time than when the user intended or a macro statement executes at an unexpected time. A prime example of the importance of timing is when you use CALL SYMPUT to write a DATA step variable to a macro variable. You cannot use this macro variable in the same DATA step where it is defined; only in subsequent steps (after the DATA step's RUN statement).

The key to preventing timing errors is to understand how the macro processor works. In simplest terms, the two major steps are compilation and execution. The compilation step resolves all macro code to compiled code. Then the code is executed. Most timing errors occur because of the following:

- the user expects something to happen during compilation that does not actually occur until execution
- expects something to happen later but is actually executed right away

Here are two examples to help you understand why the timing of compilation and execution can be important.

Example of a Macro Statement Executing Immediately

In the following program, the user intends to use the %LET statement and the ECONOMICAL macro variable to indicate whether vehicles in data set Sashelp.Cars meet or exceed a specific average MPG:

```
data economical;
   set sashelp.cars;
   avg_mpg = (MPG_Highway + MPG_City)/2;
   if (avg_mpg >= 35) then
   do;
      %let economical = Yes;  /* ERROR */
      output;
   end;
run;

%put &economical;
```

However, the results differ from the user's expectations. The %LET statement is executed immediately and the DATA step is being compiled--before the data set is read. Therefore, the %LET statement executes regardless of the results of the IF condition. Even if the data set contains no observations where AVG_MPG is greater than or equal to 35, ECONOMICAL is always `Yes`.

The solution is to set the macro variable's value by a means that is controlled by the IF logic and does not execute unless the IF statement is true. In this case, the user should use CALL SYMPUT, as in the following correct program:

```
%let economical = No;
data economical;
   set sashelp.cars;
   avg_mpg = (MPG_Highway + MPG_City)/2;
   if (avg_mpg >= 35) then
   do;
      call symput("economical", "Yes");
      output;
   end;
```

```
run;

%put &economical;
```

When this program is submitted, the value of ECONOMICAL is set to `Yes` only if at lease one observation is found with AVG_MPG greater than or equal to 35. Note that the variable was initialized to `No`. It is generally a good idea to initialize your macro variables.

Resolving Macro Resolution Problems Occurring during DATA Step Compilation

In the previous example, you learned you had to use CALL SYMPUT to conditionally assign a macro variable a value in a DATA step. So, you submit the following program:

```
%let best_mpg = 0;
data economical;
   set sashelp.cars end=last;
   retain best_mpg 0;
   avg_mpg = (MPG_Highway + MPG_City)/2;
   if (avg_mpg >= 35) then do;
      if (best_mpg < avg_mpg) then best_mpg = avg_mpg;
      output;
   end;
   if (last) then do;
      call symput("best_mpg", strip(put(best_mpg,5.)));
      put "The best average MPG is &best_mpg..";
   end;
run;
```

Note: Use double quotation marks in statements like PUT, because macro variables do not resolve when enclosed in single quotation marks.

If BEST_MPG was 63, you would expect to see a log message like the following:

```
The best average MPG is 63.
```

However, no matter what BEST_MPG is, the following message is sent to the log:

```
The best average MPG is 0.
```

When the DATA step is being compiled, &BEST_MPG is sent to the macro facility for resolution, and the result is passed back before the DATA step executes. To achieve the desired result, submit this corrected program instead:

```
%let best_mpg = 0;
data economical;
   set sashelp.cars end=last;
   retain best_mpg 0;
   avg_mpg = (MPG_Highway + MPG_City)/2;
   if (avg_mpg >= 35) then do;
      if (best_mpg < avg_mpg) then best_mpg = avg_mpg;
```

```
         output;
      end;
      if (last) then do;
         call symput("best_mpg", strip(put(best_mpg,5.)));
      end;
   run;

   %put The best average MPG is &best_mpg..;
```

Here is another example of erroneously referring to a macro variable in the same step that creates it:

```
   data _null_;
      retain total 0;
      set sashelp.prdsale end=final;
      total = total + actual;
      call symput("macvar",put(total,dollar14.2));
      if final then put "*** total=&macvar ***"; /* ERROR */
   run;
```

When these statements are submitted, the following lines are written to the SAS log:

```
WARNING: Apparent symbolic reference MACVAR not resolved.

*** total=&macvar ***
```

As this DATA step is tokenized and compiled, the `&`causes the word scanner to trigger the macro processor, which looks for a MACVAR entry in a symbol table. Because such an entry does not exist, the macro processor generates the warning message. Because the tokens remain on the input stack, they are transferred to the DATA step compiler. During DATA step execution, the CALL SYMPUT statement creates the macro variable MACVAR and assigns a value to it. However, the text `&macvar` in the PUT statement occurs because the text has already been processed while the macro was being compiled. If you were to resubmit these statements, the macro would appear to work correctly, but the value of MACVAR would reflect the value set during the previous execution of the DATA step. This value can be misleading.

Remember that in general, the `%` and `&`trigger immediate execution or resolution during the compilation stage of the rest of your SAS code.

For more examples and explanation of how CALL SYMPUT creates macro variables, see "Special Cases of Scope with the CALL SYMPUT Routine" on page 77.

Solving Problems with the Autocall Facility

The autocall facility is an efficient way of storing and using production (debugged) macros. When a call to an autocall macro produces an error, the cause is one of two things:

- an erroneous autocall library specification

- an invalid autocall macro definition

If the error is the autocall library specification and the MERROR option is set, SAS can generate any or all of the following warnings:

```
WARNING: No logical assign for filename
FILENAME.
WARNING: Source level autocall is not found or cannot be opened.
         Autocall has been suspended and OPTION NOMAUTOSOURCE has
         been set. To use the autocall facility again, set OPTION
         MAUTOSOURCE.
WARNING: Apparent invocation of macro
MACRO-NAME not resolved.
```

If the error is in the autocall macro definition, SAS generates a message like the following:

```
NOTE: Line generated by the invoked macro
"MACRO-NAME".
```

Fixing Autocall Library Specifications

When an autocall library specification causes an error, it is because the macro processor cannot find the member containing the autocall macro definition in the library or libraries specified in the SASAUTOS system option.

To correct this error, follow these steps.

1. If the unresolved macro call created an invalid SAS statement, submit a single semicolon to terminate the invalid statement. SAS is then able to correctly recognize subsequent statements.

2. Look at the value of the SASAUTOS system option by printing the output of the OPTIONS procedure or by viewing the OPTIONS window in the SAS windowing environment. (Or, edit your SAS configuration file or SAS autoexec file.) Verify each fileref or directory name. If you find an error, submit a new OPTIONS statement or change the SASAUTOS setting in the OPTIONS window.

3. Check the MAUTOSOURCE system option. If SAS could not open at least one library, it sets the NOMAUTOSOURCE option. If NOMAUTOSOURCE is present, reset MAUTOSOURCE with a new OPTIONS statement or the OPTIONS window.

4. If the library specifications are correct, check the contents of each directory to verify that the autocall library member exists and that it contains a macro definition of the same name. If the member is missing, add it.

5. Set the MRECALL option with a new OPTIONS statement or the OPTIONS window. By default, the macro processor searches only once for an undefined macro. Setting this option causes the macro processor to search the autocall libraries for the specification again.

6. Call the autocall macro, which includes and submits the autocall macro source.

7 Reset the NOMRECALL option.

Note: Some host environments have environment variables or system-level logical names assigned to the SASAUTOS library. Check your SAS companion documentation for more information about how the SASAUTOS library specification is handled in your host environment.

Fixing Autocall Macro Definition Errors

When the autocall facility locates an autocall library member, the macro processor compiles any macros in that library member. It stores the compiled macros in the catalog containing stored compiled macros. For the rest of your Compute Server session, invoking one of those macros retrieves the compiled macro from the Work library. Under no circumstances does the autocall facility use an autocall library member when a compiled macro with the same name already exists. Thus, if you invoke an autocall macro and discover you made an error when you defined it, you must correct the autocall library member for future use. Compile the corrected version directly in your program or session.

To correct an autocall macro definition in a windowing environment, do the following:

1 Use the INCLUDE command to bring the autocall library member into the SAS Code Editor window. If the macro is stored in a catalog SOURCE entry, use the COPY command to bring the program into the Code Editor window.

2 Correct the error.

3 Store a copy of the corrected macro in the autocall library with the FILE command for a macro in an external file or with a SAVE command for a macro in a catalog entry.

4 Submit the macro definition from the Code Editor window.

The macro processor then compiles the corrected version, replacing the incorrect compiled macro. The corrected, compiled macro is now ready to execute at the next invocation.

To correct an autocall macro definition in an interactive line mode session, do the following:

1 Edit the autocall macro source with a text editor.

2 Correct the error.

3 Use a %INCLUDE statement to bring the corrected library member into the Compute Server.

The macro processor then compiles the corrected version, replacing the incorrect compiled macro. The corrected, compiled macro is now ready to execute at the next invocation.

File and Macro Names for Autocall

When you want to use a macro as an autocall macro, you must store the macro in a file with the same name as the macro. Also, the file extension must be **.sas** (if your operating system uses file extensions). If you experience problems with the autocall facility, be sure the macro and filenames match and the file has the right extension when necessary.

Displaying Information about Stored Compiled Macros

To display the list of entries in a catalog containing compiled macros, you can use the Catalog window or the CATALOG procedure. The following PROC step displays the contents of a macro catalog in a SAS library identified with the libref MYSASLIB:

```
libname mysaslib 'SAS-library-path';
proc catalog catalog=mysaslib.sasmacr;
   contents;
run;
quit;
```

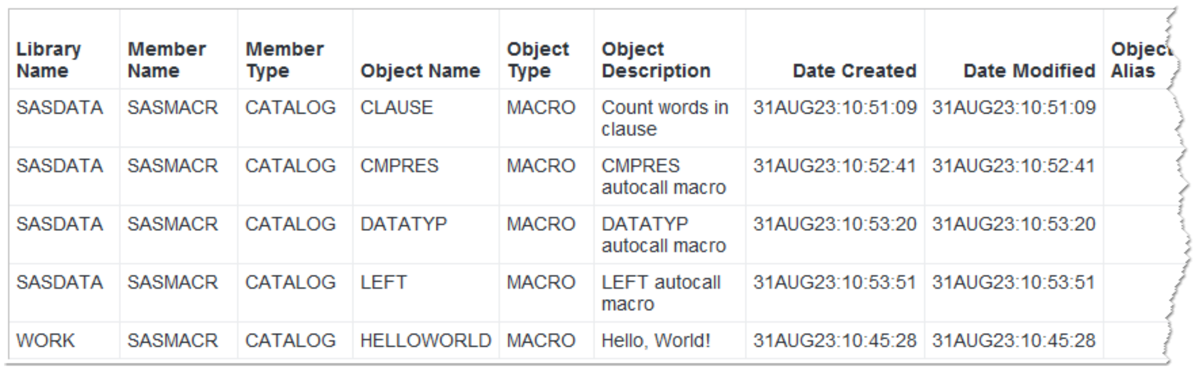
You can also use PROC CATALOG to display information about autocall library macros stored in SOURCE entries in a catalog. You cannot use PROC CATALOG or the Explorer window to copy, delete, or rename stored compiled macros.

You can use the MCOMPILENOTE system option to issue a note to the log upon the completion of the compilation of any macro. For more information, see "MCOMPILENOTE= System Option" on page 444.

You can use PROC SQL to retrieve information about all compiled macros. For example, submitting these statements produces output similar to the following output:

```
   proc sql;
      select * from dictionary.catalogs
         where memname in ('SASMACR');
   run;
   quit;
```

Output 10.1 *Sample Output from PROC SQL Program for Viewing Compiled Macros*



Library Name	Member Name	Member Type	Object Name	Object Type	Object Description	Date Created	Date Modified	Objec Alias
SASDATA	SASMACR	CATALOG	CLAUSE	MACRO	Count words in clause	31AUG23:10:51:09	31AUG23:10:51:09	
SASDATA	SASMACR	CATALOG	CMPRES	MACRO	CMPRES autocall macro	31AUG23:10:52:41	31AUG23:10:52:41	
SASDATA	SASMACR	CATALOG	DATATYP	MACRO	DATATYP autocall macro	31AUG23:10:53:20	31AUG23:10:53:20	
SASDATA	SASMACR	CATALOG	LEFT	MACRO	LEFT autocall macro	31AUG23:10:53:51	31AUG23:10:53:51	
WORK	SASMACR	CATALOG	HELLOWORLD	MACRO	Hello, World!	31AUG23:10:45:28	31AUG23:10:45:28	

To display information about compiled macros when you invoke them, use the SAS system options MLOGIC, MPRINT, and SYMBOLGEN. When you specify the SAS system option MLOGIC, the libref and date of compilation of a stored compiled macro are written to the log along with the usual information displayed during macro execution.

Solving Problems with Expression Evaluation

The following macro statements use the %EVAL function:

Table 10.3 *Macro Statements That Use the %EVAL Function*

%DO	%IF-%THEN	%SCAN
%DO %UNTIL	%QSCAN	%SYSEVALF
%DO %WHILE	%QSUBSTR	%SUBSTR

In addition, you can use the %EVAL function to specify an expression evaluation.

The most common errors that occur while evaluating expressions are the presence of character operands where numeric operands are required or ambiguity about whether a token is a numeric operator or a character value. Chapter 6, “Macro Expressions,” on page 85 discusses these and other macro expression errors.

Quite often, an error occurs when a special character or a keyword appears in a character string. Consider the following program:

```
%macro conjunct(word);
   %if &word = and or &word = but or &word = or %then   /* ERROR */
      %do;
         %put *** &word is a conjunction. ***;
      %end;
   %else
      %do;
         %put *** &word is not a conjunction. ***;
      %end;
```

```
%mend conjunct;

%conjunct(and);
```

In the %IF statement, the values of WORD being tested are ambiguous — they could also be interpreted as the numeric operators AND and OR. Therefore, SAS generates the following error messages in the log:

```
ERROR: A character operand was found in the %EVAL function or %IF
       condition where a numeric operand is required. The condition
       was:word = and or      &word = but or      &word = or
ERROR: The macro will stop executing.
```

To fix this problem, use the quoting functions %BQUOTE and %STR, as in the following corrected program:

```
%macro conjunct(word);
   %if %bquote(&word) = %str(and) or %bquote(&word) = but or
          %bquote(&word) = %str(or) %then
      %do;
         %put *** &word is a conjunction. ***;
      %end;
   %else
      %do;
         %put *** &word is not a conjunction. ***;
      %end;
%mend conjunct;

%conjunct(and);
```

In the corrected program, the %BQUOTE function quotes the result of the macro variable resolution (in case the user passes in a word containing an unmatched quotation mark or some other odd value). The %STR function quotes the comparison values AND and OR at compile time, so they are not ambiguous. You do not need to use %STR on the value BUT, because it is not ambiguous (not part of the SAS or macro languages).

Here is the result:

```
*** and is a conjunction. ***
```

For more information, see Chapter 7, “Macro Quoting,” on page 95.

Debugging Techniques

Using System Options to Track Problems

The SAS system options MLOGIC, MLOGICNEST, MPRINT, MPRINTNEST, and SYMBOLGEN can help you track the macro code and SAS code generated by your macro. Messages generated by these options appear in the SAS log, prefixed by the name of the option responsible for the message.

Note: Whenever you use the macro facility, use the following macro options: MACRO, MERROR, and SERROR. SOURCE is a system option that is helpful when using the macro facility. It is also helpful to use the SOURCE2 system option when using the %INCLUDE.

Although the following sections discuss each system option separately, you can, of course, combine them. However, each option can produce a significant amount of output, and too much information can be as confusing as too little. So, use only those options that you think you might need and turn them off when you complete the debugging.

Tracing the Flow of Execution with MLOGIC

The MLOGIC system option traces the flow of execution of your macro, including the resolution of parameters, the scope of variables (global or local), the conditions of macro expressions being evaluated, the number of loop iterations, and the beginning and end of each macro execution. Use the MLOGIC option when you think a bug lies in the program logic (as opposed to simple syntax errors).

Note: MLOGIC can produce a lot of output, so use it only when necessary, and turn it off when debugging is finished.

In the following example, the macro FIRST calls the macro SECOND to evaluate an expression:

```
%macro second(param);
   %let a = %eval(&param); &a
%mend second;

%macro first(exp);
   %if (%second(&exp) ge 0) %then
      %put **** result >= 0 ****;
   %else
      %put **** result < 0 ****;
%mend first;

options mlogic;
%first(1+2)
options nomlogic;
```

Submitting this example with option MLOGIC shows when each macro starts execution, the values of passed parameters, and the result of the expression evaluation.

```
MLOGIC(FIRST):  Beginning execution.
MLOGIC(FIRST):  Parameter EXP has value 1+2
MLOGIC(SECOND):  Beginning execution.
MLOGIC(SECOND):  Parameter PARAM has value 1+2
MLOGIC(SECOND):  %LET (variable name is A)
MLOGIC(SECOND):  Ending execution.
MLOGIC(FIRST):  %IF condition (%second(&exp) ge 0) is TRUE
MLOGIC(FIRST):  %PUT **** result >= 0 ****
**** result >= 0 ****
MLOGIC(FIRST):  Ending execution.
```

Nesting Information Generated by MLOGICNEST

MLOGICNEST allows the macro nesting information to be written to the SAS log in the MLOGIC output. The setting of MLOGICNEST does not imply the setting of MLOGIC. You must set both MLOGIC and MLOGICNEST in order for output (with nesting information) to be written to the SAS log.

For more information and an example, see "MLOGICNEST System Option" on page 461.

Examining the Generated SAS Statements with MPRINT

The MPRINT system option writes to the SAS log each SAS statement generated by a macro. Use the MPRINT option when you suspect your bug lies in code that is generated in a manner that you did not expect.

For example, the following program generates a simple DATA step:

```
%macro second(param);
   %let a = %eval(&param); &a
%mend second;

%macro first(exp);
   data _null_;
      var=%second(&exp);
      put var=;
   run;
%mend first;

options mprint;
%first(1+2)
options nomprint;
```

When you submit these statements with option MPRINT, these lines are written to the SAS log:

```
MPRINT(FIRST):   DATA _NULL_;
MPRINT(FIRST):   VAR=
MPRINT(SECOND):  3
MPRINT(FIRST):  ;
MPRINT(FIRST):   PUT VAR=;
MPRINT(FIRST):   RUN;

VAR=3
```

The MPRINT option shows you the generated text and identifies the macro that generated it.

Nesting Information Generated by MPRINTNEST

MPRINTNEST allows the macro nesting information to be written to the SAS log in the MPRINT output. This value has no effect on the MPRINT output that is sent to an external file. For more information, see “MFILE System Option” on page 452.

The setting of MPRINTNEST does not imply the setting of MPRINT. You must set both MPRINT and MPRINTNEST in order for output (with the nesting information) to be written to the SAS log.

For more information and an example, see “MPRINTNEST System Option” on page 465.

Storing MPRINT Output in an External File

You can store text that is generated by the macro facility during macro execution in an external file. Printing the statements generated during macro execution to a file is useful for debugging macros when you want to test generated text in a later session.

To use this feature, set both the MFILE and MPRINT system options on. Also assign MPRINT as the fileref for the file to contain the output generated by the macro facility:

```
options mprint mfile;
filename mprint 'external-file';
```

The external file created by the MPRINT system option remains open until the Compute Server terminates. The MPRINT text generated by the macro facility is written to the log during the SAS session and to the external file when the session ends. The text consists of program statements generated during macro execution with macro variable references and macro expressions resolved. Only statements generated by the macro are stored in the external file. Any program statements outside the macro are not written to the external file. Each statement begins on a new line with one space separating words. The text is stored in the external file without the `MPRINT(`*macroname*`)`: prefix, which is displayed in the log.

If MPRINT is not assigned as a fileref or if the file cannot be accessed, warnings are written to the log and MFILE is turned off. To use the feature again, you must specify MFILE again.

By default, the MPRINT and MFILE options are off.

The following example uses the MPRINT and MFILE options to store generated text in the external file named TempOut:

```
options mprint mfile;
filename mprint 'TEMPOUT';

%macro temp;
   data one;
      %do i=1 %to 3;
         x&i=&i;
      %end;
   run;
%mend temp;

%temp

options nomprint nomfile;
```

The macro facility writes the following lines to the SAS log and creates the external file named TempOut:

```
MPRINT(TEMP):   DATA ONE;
NOTE: The macro generated output from MPRINT will also be written
      to external file '/u/local/abcdef/TEMPOUT' while OPTIONS
      MPRINT and MFILE are set.
MPRINT(TEMP):   X1=1;
MPRINT(TEMP):   X2=2;
MPRINT(TEMP):   X3=3;
MPRINT(TEMP):   RUN;
```

When the SAS session ends, the file TempOut contains:

```
DATA ONE;
X1=1;
X2=2;
X3=3;
RUN;
```

Note: Using MPRINT to write code to an external file is a debugging tool only. It should not be used to create SAS code files for purposes other than debugging.

Examining Macro Variable Resolution with SYMBOLGEN

The SYMBOLGEN system option tells you what each macro variable resolves to by writing messages to the SAS log. This option is especially useful in spotting quoting problems, where the macro variable resolves to something other than what you intended because of a special character.

For example, suppose you submit the following statements:

```
options symbolgen;

%let a1=dog;
%let b2=cat;
%let b=1;
%let c=2;
%let d=a;
%let e=b;
%put **** &&&d&b ****;
%put **** &&&e&c ****;

options nosymbolgen;
```

The SYMBOLGEN option writes these lines to the SAS log:

```
SYMBOLGEN:  && resolves to &.
SYMBOLGEN:  Macro variable D resolves to a
SYMBOLGEN:  Macro variable B resolves to 1
SYMBOLGEN:  Macro variable A1 resolves to dog
**** dog ****

SYMBOLGEN:  && resolves to &.
SYMBOLGEN:  Macro variable E resolves to b
SYMBOLGEN:  Macro variable C resolves to 2
SYMBOLGEN:  Macro variable B2 resolves to cat
**** cat ****
```

Reading the log provided by the SYMBOLGEN option is easier than examining the program statements to trace the indirect resolution. Notice that the SYMBOLGEN option traces each step of the macro variable resolution by the macro processor. When the resolution is complete, the %PUT statement writes the value to the SAS log.

When you use SYMBOLGEN to trace the values of macro variables that have been masked with a macro quoting function, you might see an additional message about the quoting being “stripped for printing.” For example, suppose you submit the following statements, with SYMBOLGEN:

```
options symbolgen;
%let nickname = %str(My name%'s O%'Malley, but I%'m called Bruce);
%put *** &nickname ***;
options nosymbolgen;
```

The SAS log contains the following after these statements have executed:

```
SYMBOLGEN:  Macro variable NICKNAME resolves to
                            My name's O'Malley, but I'm called Bruce
SYMBOLGEN:  Some characters in the above value which were
                           subject to macro quoting have been
                           unquoted for printing.
*** My name's O'Malley, but I'm called Bruce ***
```

You can ignore the unquoting message.

Using the %PUT Statement to Track Problems

Along with using the SYMBOLGEN system option to write the values of macro variables to the SAS log, you might find it useful to use the %PUT statement while developing and debugging your macros. When the macro is finished, you can delete or comment out the %PUT statements. The following table provides some occasions where you might find the %PUT statement helpful in debugging, and an example of each:

Table 10.4 *Example %PUT Statements That Are Useful When Debugging Macros*

Situation	Example
Show a macro variable's value	`%PUT ****&=`*`variable-name`*`****;`
Check leading or trailing blanks in a variable's value	`%PUT ***&`*`variable-name`*`***;`
Check double-ampersand resolution, as during a loop	`%PUT ***`*`variable-name&i`*` = &&`*`variable-name`*`***;`
Check evaluation of a condition	`%PUT ***This condition was met.***;`

As you recall, macro variables are stored in symbol tables. There is a global symbol table, which contains global macro variables, and a local symbol table, which contains local macro variables. During the debugging process, you might find it helpful to print these tables occasionally to examine the scope and values of a group of macro variables. To do so, use the %PUT statement with one of the following options:

ALL
: describes all currently defined macro variables, regardless of scope. User-generated global and local variables as well as automatic macro variables are included.

AUTOMATIC
: describes all automatic macro variables. The scope is listed as AUTOMATIC. All automatic macro variables are global except SYSPBUFF. Here is an example of using _AUTOMATIC_:

```
filename cmdfile "cmdfile.sas";
data _null_;
   file cmdfile;
   put "proc print data=sashelp.class; run; %nrstr(%put _automatic_;)";
run;

data _null_;
   %include cmdfile;
run;
```

```
filename cmdfile clear;
```

GLOBAL
: describes all global macro variables that were not created by the macro processor. The scope is listed as GLOBAL. Automatic macro variables are not listed.

LOCAL
: describes user-generated local macro variables defined within the currently executing macro. The scope is listed as the name of the macro in which the macro variable is defined.

USER
: describes all user-generated macro variables, regardless of scope. For global macro variables, the scope is GLOBAL; for local macro variables, the scope is the name of the macro.

The following example uses the %PUT statement with the argument _USER_ to examine the global and local variables available to the macro TOTINV. Notice the use of the user-generated macro variable TRACE to control when the %PUT statement writes values to the log. See Example Code 10.1 on page 151.

```
%macro totinv(var);
   %global macvar;
   data inv;
      retain total 0;
      set Sasuser.Houses end=final;
      total=total+&var;
      if final then call symput("macvar",put(total,dollar14.2));
   run;

   %if &trace = ON  %then
      %do;
         %put *** Tracing macro scopes. ***;
         %put _USER_;
      %end;
%mend totinv;

%let trace=ON;
%totinv(price)
%put *** TOTAL=&macvar ***;
```

The first %PUT statement in the macro TOTINV writes the message about tracing being on and then writes the scope and value of all user-generated macro variables to the SAS log.

```
*** Tracing macro scopes. ***
TOTINV VAR price
GLOBAL TRACE ON
GLOBAL MACVAR  $1,241,100.00
*** TOTAL= $1,241,100.00 ***
```

For a more detailed discussion of macro variable scopes, see Chapter 11, “Writing Efficient and Portable Macros,” on page 171.

11

Writing Efficient and Portable Macros

Writing Efficient and Portable Macros

The macro facility is a powerful tool for making your SAS code development more efficient. But macros are only as efficient as you make them. There are several techniques and considerations for writing efficient macros. You can extend the power of the macro facility by creating macros that can be used on more than one host environment. In order to do this, there are additional considerations for writing portable macros.

Keeping Efficiency in Perspective

Efficiency is an elusive thing, hard to quantify and harder still to define. What works with one application might not work with another, and what is efficient on one host environment might be inefficient on a different system. However, there are some generalities that you should keep in mind.

Usually, efficiency issues are discussed in terms of CPU cycles, elapsed time, I/O hits, memory usage, disk storage, and so on. This section does not give benchmarks in these terms because of all the variables involved. A program that runs only once needs different tuning than a program that runs hundreds of times. An application running on a mainframe has different hardware parameters than an application developed on a desktop PC. You must keep efficiency in perspective with your environment.

There are different approaches to efficiency, depending on what resources you want to conserve. Are CPU cycles more critical than I/O hits? Do you have lots of memory but no disk space? Taking stock of your situation before deciding how to tune your programs is a good idea.

The area of efficiency most affected by the SAS macro facility is human efficiency — how much time is required to both develop and maintain a program. Autocall macros are particularly important in this area because the autocall facility provides code reusability. Once you develop a macro that performs a task, you can save it and use it for the following:

- in the application that you developed it for
- in future applications without any further work

A library of reusable, immediately callable macros is a boon to any application development team.

The stored compiled macro facility (described in Chapter 9, “Storing and Reusing Macros,” on page 137) might reduce execution time by enabling previously compiled macros to be accessed during different SAS jobs and sessions. But it is a tool that is efficient only for production applications, not during application development. So the efficiency techniques that you choose depend not only on your hardware and personnel situation, but also on the stage that you have reached in your application development process.

Also, remember that incorporating macro code into a SAS application does not automatically make the application more efficient. When designing a SAS application, concentrate on making the basic SAS code that macros generate more efficient.

Writing Efficient Macros

Use Macros Wisely

An application that uses a macro to generate only constant text can be inefficient. In general, for these situations consider using a %INCLUDE statement. The %INCLUDE statement does not have to compile the code first (it is executed immediately). Therefore, it might be more efficient than using a macro (especially if the code is executed only once). For more information, see “%INCLUDE Statement” in *SAS Global Statements: Reference*.

If you use the same code repeatedly, it might be more efficient to use a macro. A macro is compiled only once during a SAS job, no matter how many times it is called.

Use Name Style Macros

Macros come in three invocation types: name style, command style, and statement style. Of the three, name style is the most efficient because name style macros always begin with a `%`, which immediately tells the word scanner to pass the token to the macro processor. With the other two types, the word scanner does not know immediately whether the token should be sent to the macro processor. Therefore, time is wasted while the word scanner determines whether the token should be sent.

Avoid Nested Macro Definitions

Nesting macro definitions inside other macros is usually unnecessary and inefficient. When you call a macro that contains a nested macro definition, the macro processor generates the nested macro definition as text and places it on the input stack. The word scanner then scans the definition and the macro processor compiles it. Do not nest the definition of a macro that does not change. You will cause the macro processor to compile the same macro each time that section of the outer macro is executed.

As a rule, you should define macros separately. If you want to nest a macro's scope, simply nest the macro call, not the macro definition.

For example, the macro %CLASSSTATS contains a nested macro definition for the macro %GETDATA:

```
/* Nesting a Macro Definition--INEFFICIENT */
%macro classStats(ageGroup=all);
   %macro getData;
      title "Statistics for Class";
      %let ageGroup = %sysfunc(propcase(&ageGroup));
      %if &ageGroup eq Preteen %then
         %let datasrc = sashelp.class(where=(age between 9 and 12));
      %else %if &ageGroup eq Teen %then
         %let datasrc = sashelp.class(where=(age between 13 and 19));
      %else %let datasrc = sashelp.class;

      title2 "&ageGroup Students";

      data class;
         set &datasrc;
      run;
   %mend getData;

   %getData;

   proc means data=class;
   run;
%mend classStats;

%classStats;
%classStats(agegroup = preteen);
%classStats(agegroup = teen);
```

Each time the macro %CLASSSTATS is called, the macro processor generates the definition of the macro %GETDATA as text, recognizes a macro definition, and then compiles the macro %GETDATA. In this case, %CLASSSTATS was called three times, which means the %GETDATA macro was compiled three times. With relatively few statements, this task takes only micro-seconds; but in large macros with hundreds of statements, the wasted time could be significant.

The value of AGEGROUP is available to %GETDATA because its call is within the definition of %CLASSSTATS. Therefore, it is unnecessary to nest the definition of %GETDATA to make values available to %GETDATA's scope. Nesting definitions is also unnecessary because no values in the definition of the %GETDATA statement are dependent on values that change during the execution of %CLASSSTATS. (Even if the definition of the %GETDATA statement depended on such values, you could use a global macro variable to effect the changes, rather than nest the definition.)

The following program shows the macros defined separately:

```
/* Separating Macro Definitions--EFFICIENT */
%macro getData;
   title "Statistics for Class";
   %let ageGroup = %sysfunc(propcase(&ageGroup));
   %if &ageGroup eq Preteen %then
      %let datasrc = sashelp.class(where=(age between 9 and 12));
   %else %if &ageGroup eq Teen %then
      %let datasrc = sashelp.class(where=(age between 13 and 19));
   %else %let datasrc = sashelp.class;

   title2 "&ageGroup Students";
   data class;
      set &datasrc;
   run;
%mend getData;

%macro classStats(ageGroup=all);
   %getData;

   proc means data=class;
   run;
%mend classStats;

%classStats;
%classStats(agegroup = preteen);
%classStats(agegroup = teen);
title;
```

Here, because the definition of the macro %GETDATA is outside the definition of the macro %CLASSSTATS, %GETDATA is compiled only once, even though %CLASSSTATSis called three times. Again, the value of AGEGROUP is available to %GETDATA because its call is within the definition of %CLASSSTATS.

TIP Another reason to define macros separately is because it makes them easier to maintain, each in a separate file.

Assign Function Results to Macro Variables

It is more efficient to resolve a variable reference than it is to evaluate a function. Therefore, assign the results of frequently used functions to macro variables.

For example, the following macro is inefficient because the length of the macro variable THETEXT must be evaluated at every iteration of the %DO %WHILE statement:

```
/* INEFFICIENT MACRO */
%macro test(thetext);
   %let x=1;
      %do %while (&x > %length(&thetext));
         .
         .
         .
      %end;
%mend test;

%test(Four Score and Seven Years Ago)
```

A more efficient method would be to evaluate the length of THETEXT once and assign that value to another macro variable. Then, use that variable in the %DO %WHILE statement, as in the following program:

```
/* MORE EFFICIENT MACRO */
```

```
%macro test2(thetext);
   %let x=1;
   %let length=%length(&thetext);
   %do %while (&x > &length);
      .
      .
      .
   %end;
%mend test2;

%test(Four Score and Seven Years Ago)
```

As another example, suppose you want to use the %SUBSTR function to pull the year out of the value of SYSDATE. Instead of using %SUBSTR repeatedly in your code, assign the value of the %SUBSTR(&SYSDATE, 6) to a macro variable, and use that variable whenever you need the year.

Turn Off System Options When Appropriate

While the debugging system options, such as MPRINT and MLOGIC, are very helpful at times, it is inefficient to run production (debugged) macros with this type of system option set to on. For production macros, run your job with the following settings: NOMLOGIC, NOMPRINT, NOMRECALL, and NOSYMBOLGEN.

Even if your job has no errors, if you run it with these options turned on you incur the overhead that the options require. By turning them off, your program runs more efficiently.

Note: Another approach to deciding when to use MPRINT versus NOMPRINT is to match this option's setting with the setting of the SOURCE option. That is, if your program uses the SOURCE option, it should also use MPRINT. If your program uses NOSOURCE, then run it with NOMPRINT as well.

Note: If you do not use autocall macros, use the NOMAUTOSOURCE system option. If you do not use stored compiled macros, use the NOMSTORED system option.

Use the Stored Compiled Macro Facility

The stored compiled macro facility reduces execution time by enabling macros compiled in a previous SAS job or session to be accessed during subsequent SAS jobs and sessions. Therefore, these macros do not need to be recompiled. Use the stored compiled macro facility only for production (debugged) macros. It is not efficient to use this facility when developing a macro application.

CAUTION
Save the source code. You cannot re-create the source code for a macro from the compiled code. You should keep a copy of the source code in a safe place, in case the compiled code becomes corrupted for some reason. Having a copy of the source is also necessary if you intend to modify the macro at a later time.

For more information about the stored compiled macro facility, see Chapter 9, "Storing and Reusing Macros," on page 137.

Note: The compiled code generated by the stored compiled macro facility is not portable. If you need to transfer macros to another host environment, you must move the source code and recompile and store it on the new host.

Centrally Store Autocall Macros

When using the autocall facility, it is most efficient in terms of I/O to store all your autocall macros in one library and append that library name to the beginning of the SASAUTOS system option specification. Of course, you could store the autocall macros in as many libraries as you want. But each time you call a macro, each library is searched sequentially until the macro is found. Opening and searching only one library reduces the time SAS spends looking for macros.

However, it might make more sense, if you have hundreds of autocall macros, to have them separated into logical divisions according to the following:

- purpose
- levels of production
- who supports them
- and so on

As usual, you must balance reduced I/O against ease-of-use and ease-of-maintenance.

All autocall libraries in the concatenated list are opened and left open during a SAS job or session. The first time you call an autocall macro, any library that did not open the first time is tested again each time an autocall macro is used. Therefore, it is extremely inefficient to have invalid pathnames in your SASAUTOS system option specification. You see no warnings about this wasted effort on the part of SAS, unless no libraries at all open.

There are two efficiency tips involving the autocall facility:

- Do not store nonmacro code in autocall library files.
- Do not store more than one macro in each autocall library file.

Although these two practices are used by SAS and do work, they contribute significantly to code-maintenance effort and therefore are less efficient.

Other Useful Efficiency Tips

Here are some other efficiency techniques that you can try:

- Reset macro variables to null if the variables are no longer going to be referenced.
- Use triple ampersands to force an additional scan of macro variables with long values, when appropriate. For more information, see "Storing Only One Copy of a Long Macro Variable Value" on page 178.
- Adjust the values of "MSYMTABMAX= System Option" on page 469 and "MVARSIZE= System Option" on page 471 to fit your situation. In general, increase the values if disk space is in short supply; decrease the values if memory is in short supply. MSYMTABMAX affects the space available for storing macro variable symbol tables; MVARSIZE affects the space available for storing values of individual macro variables.

Storing Only One Copy of a Long Macro Variable Value

Because macro variables can have very long values, the way you store macro variables can affect the efficiency of a program. Indirect references using three ampersands enable you to store fewer copies of a long value.

For example, suppose your program contains long macro variable values that represent sections of SAS programs:

```
%let pgm=%str(data class;
   set sashelp.class;
   where age >= 15;

   proc print;
      var name age sex;
);
```

You want the SAS program to end with a RUN statement:

```
%macro check(val);
   /* first version */
   &val
   %if %index(&val,%str(run;))=0 %then %str(run;);
%mend check;
```

First, the macro CHECK generates the program statements contained in the parameter VAL (a macro variable that is defined in the %MACRO statement and passed in from the macro call). Then, the %INDEX function searches the value of VAL for the characters `run;`. (The %STR function causes the semicolon to be treated as text.) If the characters are not present, the %INDEX function returns 0.

The %IF condition becomes true, and the macro processor generates a RUN statement.

To use the macro CHECK with the variable PGM, assign the parameter VAL the value of PGM in the macro call:

```
   %check(&pgm)
```

As a result, SAS sees the following statements:

```
data class;
   set sashelp.class;
   where age >= 15;

   proc print;
      var name age sex;
run;
```

The macro CHECK works properly. However, the macro processor assigns the value of PGM as the value of VAL during the execution of CHECK. Thus, the macro processor must store two long values (the value of PGM and the value of VAL) while CHECK is executing.

To make the program more efficient, write the macro so that it uses the value of PGM rather than copying the value into VAL:

```
%macro check2(val);
   /* more efficient macro */
   &&&val
   %if %index(&&&val,%str(run;))=0 %then %str(run;);
%mend check2;

%check2(pgm)
```

The macro CHECK2 produces the same result as the macro CHECK:

```
data class;
   set sashelp.class;
   where age >= 15;

   proc print;
      var name age sex;
run;
```

However, in the macro CHECK2, the value assigned to VAL is simply the name `PGM`, not the value of PGM. The macro processor resolves &&&VAL into &PGM and then into the SAS statements contained in the macro variable PGM. Thus, the long value is stored only once.

Writing Portable Macros

Using Portable SAS Language Functions with %SYSFUNC

If your code runs in two different environments, you have essentially doubled the worth of your development effort. But portable applications require some planning ahead. For more details about any host-specific feature of SAS, see the SAS documentation for your host environment.

You can use the %SYSFUNC macro function to access SAS language functions to perform most host-specific operations, such as opening or deleting a file. For more information, see “%SYSFUNC Macro Function” on page 356.

Using %SYSFUNC to access portable SAS language functions can save you a lot of macro coding (and is therefore not only portable but also more efficient). The following table lists some common host-specific tasks and the functions that perform those tasks.

Table 11.1 *Portable SAS Language Functions and Their Uses*

Task	SAS Language Function or Functions
Assign and verify existence of fileref and physical file	FILENAME, FILEREF, PATHNAME
Open a file	FOPEN, MOPEN
Verify existence of a file	FEXIST, FILEEXIST
Get information about a file	FINFO, FOPTNAME, FOPTNUM
Write data to a file	FAPPEND, FWRITE
Read from a file	FPOINT, FREAD, FREWIND, FRLEN
Close a file	FCLOSE

Task	SAS Language Function or Functions
Delete a file	FDELETE
Open a directory	DOPEN
Return information about a directory	DINFO, DNUM, DOPTNAME, DOPTNUM, DREAD
Close a directory	DCLOSE
Read a host-specific option	GETOPTION
Interact with the File Data Buffer (FDB)	FCOL, FGET, FNOTE, FPOS, FPUT, FSEP
Assign and verify librefs	LIBNAME, LIBREF, PATHNAME
Get information about executed host environment commands	SYSRC

Note: Of course, you can also use other functions, such as ABS, MAX, and TRANWRD, with %SYSFUNC. A few SAS language functions are not available with %SYSFUNC. For more information, see "%SYSFUNC Macro Function" on page 356.

Example Using %SYSFUNC

The following program deletes the file identified by the fileref MyFile:

```
%macro testfile(filrf);
   %let
rc=%sysfunc(filename(filrf,physical-filename));
   %if &rc = 0 and %sysfunc(fexist(&filrf)) %then
      %let rc=%sysfunc(fdelete(&filrf));
   %let rc=%sysfunc(filename(filrf));
%mend testfile;

%testfile(myfile)
```

Using Automatic Variables with Host-Specific Values

Macro Variables by Task

The automatic macro variables are available under all host environments, but the values are determined by each host. The following table lists the macro variables by task. The “Type” column tells you if the variable can be changed (Read and Write) or can be inspected (Read Only).

Table 11.2 Automatic Macro Variables with Host-Specific Results

Task	Automatic Macro Variable	Type
List the name of the current graphics device on DEVICE=.	SYSDEVIC	Read and write
List of the mode of execution (values are FORE or BACK). Some host environments allow only one mode, FORE.	SYSENV	Read-only
List the name of the currently executing batch job, user ID, or process. For example, on Linux, SYSJOBID is the PID.	SYSJOBID	Read-only
List the last return code generated by your host environment, based on commands executed using the X statement in open code. The X command in the SAS windowing environment, or the %SYSEXEC (or %TSO or %CMS) macro statements. The default value is 0.	SYSRC	Read and write
List the abbreviation of the host environment that you are using.	SYSSCP	Read-only
List a more detailed abbreviation of the host environment that you are using.	SYSSCPL	Read-only

Task	Automatic Macro Variable	Type
Retrieve a character string that was passed to SAS by the SYSPARM= system option.	SYSPARM	Read and write
Time zone name based on TIMEZONE option.	SYSTIMEZONE	Read-only
Time zone ID based on TIMEZONE option.	SYSTIMEZONEIDENT	Read-only
Current time zone offset based on TIMEZONE option.	SYSTIMEZONEOFFSET	Read-only

Examples Using SYSSCP and SYSSCPL

The macro DELFILE uses the value of SYSSCP to determine the platform that is running SAS and deletes a TMP file. FILEREF is a macro parameter that contains a file name. Because the file name is host-specific, making it a macro parameter enables the macro to use whatever file name syntax is necessary for the host environment.

```
%macro delfile(fileref);
    /* Linux */
   %if &sysscp=HP 800 or &sysscp=HP 300 %then %do;
         X "rm &fileref..TMP";
   %end;

%mend delfile;
```

Here is a call to the macro DELFILE in a environment that deletes a file named /home/*userid*/Doc1.Tmp:

```
%delfile(/home/userid/Doc1)
```

In this program, note the use of the portable %SYSEXEC statement to carry out the host-specific operating system commands.

Now, suppose you know your macro application is going to run on some version of Linux. The SYSSCPL automatic macro variable provides information about the name of the host environment, similar to the SYSSCP automatic macro variable. However, SYSSCPL provides more information and enables you to further modify your macro code.

Example Using SYSPARM

Suppose the SYSPARM= system option is set to the name of a city. That means the SYSPARM automatic variable is set to the name of that city. You can use that value to subset a data set and generate code specific to that value. Simply by making a small change to the command that invokes SAS (or to the configuration SAS file), your SAS job will perform different tasks.

```
/* Create a data set, based on the value of the */
/* SYSPARM automatic variable. */
/* An example data set name could be MYLIB.BOSTON. */
data mylib.&sysparm;
   set mylib.alltowns;
      /* Use the SYSPARM SAS language function to */
      /* compare the value (city name) */
      /* of SYSPARM to a data set variable. */
   if town=sysparm();
run;
```

When this program executes, you end up with a data set that contains data for only the town that you are interested in. You can change what data set is generated before you start your SAS job.

Now suppose you want to further use the value of SYSPARM to control what procedures your job uses. The following macro does just that:

```
%macro select;
   %if %upcase(&sysparm) eq BOSTON %then
      %do;
         proc report ... more SAS code;
            title "Report on &sysparm";
         run;
      %end;

   %if %upcase(&sysparm) eq CHICAGO %then
      %do;
         proc chart ... more SAS code;
            title "Growth Values for &sysparm";
         run;
      %end;
   .
   .  /* more macro code */
   .
%mend select;
```

SYSPARM Details

The value of the SYSPARM automatic macro variable is the same as the value of the SYSPARM= system option, which is equivalent to the return value of the SAS language function SYSPARM. The default value is null. Because you can use the SYSPARM= system option at SAS invocation, you can set the value of the SYSPARM automatic macro variable before your Compute Server session begins.

SYSRC Details

The value of the SYSRC automatic macro variable contains the last return code generated by your host environment. The code returned is based on the following commands that you execute:

- the X statement in open code
- the X command a windowing environment
- the %SYSEXEC macro statement (as well as the nonportable %TSO and %CMS macro statements)

Use the SYSRC automatic macro variable to test the success or failure of a host environment command.

Note: Since it does not generate an error message in the SAS log, the SYSRC automatic macro variable is not useful under all host environments. For example, under some host environments, the value of this variable is always 99, regardless of the success or failure of the host environment command. Check the SAS companion for your host environment to determine whether the SYSRC automatic macro variable is useful for your host environment.

Macro Language Elements with System Dependencies

Several macro language elements are host-specific, including the following:

Any language element that relies on the sort sequence.
Examples of such expressions include %DO, %DO %UNTIL, %DO %WHILE, %IF-%THEN, and %EVAL.

For example, consider the following program:

```
%macro testsort(var);
    %if &var < a %then %put *** &var is less than a ***;
    %else %put *** &var is greater than a ***;
%mend testsort;
%testsort(1)
      /* Invoke the macro with the number 1 as the parameter. */
```

But on an ASCII system (such as Linux), the following is written to the SAS log:

```
*** 1 is less than a ***
```

MSYMTABMAX=
: The MSYMTABMAX system option specifies the maximum amount of memory available to the macro variable symbol tables. If this value is exceeded, the symbol tables are stored in a Work file on disk.

MVARSIZE=
: The MVARSIZE system option specifies the maximum number of bytes for any macro variable stored in memory. If this value is exceeded, the macro variable is stored in a Work file on disk.

%SCAN and %QSCAN
: The default delimiters that the %SCAN and %QSCAN functions use to search for words in a string are different on ASCII and EBCDIC systems. The default delimiters are

  ASCII systems
  : blank . < (+ & ! $ *) ; ^ − / , % |

  EBCDIC systems
  : blank . < (+ | & ! $ *) ; ¬ − / , % ¦ ¢

%SYSEXEC, %TSO, and %CMS
: The %SYSEXEC, %TSO, and %CMS macro statements enable you to issue a host environment command.

%SYSGET
: On some host environments, the %SYSGET function returns the value of host environment variables and symbols.

SYSPARM=
: The SYSPARM= system option can supply a value for the SYSPARM automatic macro variable at SAS invocation. It is useful in customizing a production job. For example, to create a title based on a city as part of noninteractive execution, the production program might contain the SYSPARM= system option. It can be in the SAS configuration file or the command that invokes SAS. For an example using the SYSPARM= system option in conjunction with the SYSPARM automatic macro variable, see “SYSPARM Details” on page 184.

SASMSTORE=
: The SASMSTORE= system option specifies the location of stored compiled macros.

SASAUTOS=
: The SASAUTOS= system option specifies the location of autocall macros.

Host-Specific Macro Variables

Some host environments create unique macro variables. These macro variables are not automatic macro variables. The following tables list some commonly used host-specific macro variables. Additional host-specific macro variables might be available in future releases. See your SAS companion for more details.

Table 11.3 Host-Specific Macro Variables for z/OS

Variable Name	Description
SYS99ERR	SVC99 error reason code
SYS99INF	SVC99 info reason code
SYS99MSG	YSC99 text message corresponding to the SVC error or info reason code
SYS99R15	SVC99 return code
SYSJCTID	Value of the JCTUSER field in the JCT control block
SYSJMRID	Value of the JMRUSEID field in the JCT control block
SYSUID	TSO user ID associated with the Compute Server session

Naming Macros and External Files for Use with the Autocall Facility

When naming macros that will be stored in an autocall library, there are restrictions depending on your host environment. Here is a list of some of the restrictions:

- Every host environment has file naming conventions. If the host environment uses file extensions, use **`.sas`** as the extension of your macro files.
- Although SAS names can contain underscores, some host environments do not use them in the names of external files. Some host environments that do not use underscores do use the number sign (#) and might automatically replace the # with _ when the macro is used.
- Some host environments have reserved words, such as CON and NULL. Do not use reserved words when naming autocall macros or external files.
- Some hosts have host-specific autocall macros. Do not define a macro with the same name as these autocall macros.
- Macro catalogs are not portable. Remember to always save your macro source code in a safe place.
- On Linux systems the filename that contains the autocall macro must be all lowercase letters.

12

Macro Language Elements

Macro Language Elements

The SAS macro language consists of statements, functions, and automatic macro variables. This section defines and lists these elements.

- "Macro Statements" on page 190
- "Macro Functions" on page 192
- "Automatic Macro Variables" on page 200

Also covered are the interfaces to the macro facility provided by Base SAS software, the SQL procedure, and SAS Component Language as well as selected autocall macros and macro system options.

Macro Statements

Using Macro Statements

A macro language statement instructs the macro processor to perform an operation. It consists of a string of keywords, SAS names, and special characters and operators, and it ends in a semicolon. Some macro language statements are used only in macro definitions. You can use others anywhere in a Compute Server session or job, either inside or outside macro definitions (referred to as open code). The following table lists macro language statements that you can use in both macro definitions and open code.

Table 12.1 Macro Language Statements Used in Macro Definitions and Open Code

Statement	Description
%* comment	Designates comment text.
%COPY	Copies specified items from a SAS library.
%GLOBAL	Creates macro variables that are available during the execution of an entire Compute Server session.
%IF %THEN %ELSE	Conditionally processes a portion of a macro.
%LET	Creates a macro variable and assigns it a value.
%MACRO	Begins a macro definition.
%PUT	Writes text or the values of macro variables to the SAS log.
%SYMDEL	Deletes the indicated macro variable named in the argument.
%SYSCALL	Invokes a SAS call routine.
%SYSEXEC	Issues operating system commands.

Statement	Description
%SYSLPUT	Defines a new macro variable or modifies the value of an existing macro variable on a remote host or server.
%SYSMACDELETE	Deletes a macro definition from the Work.SASMacr or Work.SASMac*n* catalog.[1]
%SYSMSTORECLEAR	Closes stored compiled macros and clears the SASMSTORE= library.
%SYSRPUT	Assigns the value of a macro variable on a remote host to a macro variable on the local host.

1 The Work.Sasmacr catalog is used to store compiled macros for the primary SAS session. In applications or programs that use side sessions, the catalog used to store compiled macros for each side session is Work.Sasmac*n*, where *n* is a unique integer.

The following table lists macro language statements that you can use only in macro definitions.

Table 12.2 *Macro Language Statements Used in Macro Definitions Only*

Statement	Description
%ABORT	Stops the macro that is executing along with the current DATA step, or Compute Server session or job.
%DO	Begins a %DO group.
%DO, Iterative	Executes statements repetitively, based on the value of an index variable.
%DO %UNTIL	Executes statements repetitively until a condition is true.
%DO %WHILE	Executes statements repetitively while a condition is true.
%END	Ends a %DO group.
%GOTO	Branches macro processing to the specified label.
%IF-%THEN/%ELSE	Conditionally processes a portion of a macro.
%label:	Identifies the destination of a %GOTO statement.
%LOCAL	Creates macro variables that are available only during the execution of the macro where they are defined.
%MEND	Ends a macro definition.

Statement	Description
%RETURN	Causes normal termination of the currently executing macro.

Macro Statements That Perform Automatic Evaluation

Some macro statements perform an operation based on an evaluation of an arithmetic or logical expression. They perform the evaluation by automatically calling the %EVAL function. If you get an error message about a problem with %EVAL when a macro does not use %EVAL only, check for one of these statements. The following macro statements perform automatic evaluation:

- %DO *macro-variable=expression* %TO *expression* <%BY *expression*>;
- %DO %WHILE(*expression*);
- %DO %UNTIL(*expression*);
- %IF *expression* %THEN *action*;

For more information about operands and operators in expressions, see Chapter 6, “Macro Expressions,” on page 85.

Macro Functions

Using Macro Functions

A macro language function processes one or more arguments and produces a result. You can use all macro functions in both macro definitions and open code. Macro functions include character functions, evaluation functions, and quoting functions. The macro language functions are listed in the following table.

Table 12.3 *Macro Functions*

Function	Description
%BQUOTE, %NRBQUOTE	Mask special characters and mnemonic operators in a resolved value at macro execution.

Function	Description
%EVAL	Evaluates arithmetic and logical expressions using integer arithmetic.
%INDEX	Returns the position of the first character of a string.
%LENGTH	Returns the length of a string.
%QUOTE, %NRQUOTE	Mask special characters and mnemonic operators in a resolved value at macro execution. Unmatched quotation marks (“ ”) and parentheses (()) must be marked with a preceding %.
%SCAN, %QSCAN	Search for a word specified by its number. %QSCAN masks special characters and mnemonic operators in its result.
%STR, %NRSTR	Mask special characters and mnemonic operators in constant text at macro compilation. Unmatched quotation marks (“ ”) and parentheses (()) must be marked with a preceding %.
%SUBSTR, %QSUBSTR	Produce a substring of a character string. %QSUBSTR masks special characters and mnemonic operators in its result.
%SUPERQ	Masks all special characters and mnemonic operators at macro execution but prevents resolution of the value.
%SYMEXIST	Returns an indication as to whether the named macro variable exists.
%SYMGLOBL	Returns an indication as to whether the named macro variable is global in scope.
%SYMLOCAL	Returns an indication as to whether the named macro variable is local in scope.
%SYSEVALF	Evaluates arithmetic and logical expressions using floating-point arithmetic.
%SYSFUNC, %QSYSFUNC	Execute SAS functions or user-written functions. %QSYSFUNC masks special characters and mnemonic operators in its result.
%SYSGET	Returns the value of a specified host environment variable.
%SYSMACEXEC	Indicates whether a macro is currently executing.

Function	Description
%SYSMACEXIST	Indicates whether there is a macro definition in the Work.SASMacr or Work.SASMacr*n* catalog.[1]
%SYSMEXECDEPTH	Returns the depth of nesting from the point of call.
%SYSMEXECNAME	Returns the name of the macro executing at a nesting level.
%SYSPROD	Reports whether a SAS software product is licensed at the site.
%UNQUOTE	Unmasks all special characters and mnemonic operators for a value.
%UPCASE, %QUPCASE	Convert characters to uppercase. %QUPCASE masks special characters and mnemonic operators in its result.

1 The Work.Sasmacr catalog is used to store compiled macros for the primary SAS session. In applications or programs that use side sessions, the catalog used to store compiled macros for each side session is Work.Sasmac*n*, where *n* is a unique integer.

Macro Character Functions

Character functions change character strings or provide information about them. The following table lists the macro character functions.

Table 12.4 Macro Character Functions

Function	Description
%INDEX	Returns the position of the first character of a string.
%LENGTH	Returns the length of a string.
%SCAN, %QSCAN	Search for a word that is specified by a number. %QSCAN masks special characters and mnemonic operators in its result.
%SUBSTR, %QSUBSTR	Produce a substring of a character string. %QSUBSTR masks special characters and mnemonic operators in its result.
%UPCASE, %QUPCASE	Convert characters to uppercase. %QUPCASE masks special characters and mnemonic operators in its result.

For macro character functions that have a Q form (for example, %SCAN and %QSCAN), the two functions work alike except that the function beginning with Q masks special characters and mnemonic operators in its result. Use the function beginning with Q when an argument has been previously masked with a macro quoting function or when you want the result to be masked (for example, when the result might contain an unmatched quotation mark or parenthesis). For more information, see “Macro Quoting” on page 96.

Many macro character functions have names corresponding to SAS character functions and perform similar tasks (such as %SUBSTR and SUBSTR). But, macro functions operate before the DATA step executes. Consider the following DATA step:

```
data out.%substr(&sysday,1,3);      /* macro function */
   set in.weekly (keep=name code sales);
   length location $4;
   location=substr(code,1,4);         /* SAS function */
run;
```

Running the program on Monday creates the data set name Out.Mon:

```
data out.MON;                       /* macro function */
   set in.weekly (keep=name code sales);
   length location $4;
   location=substr(code,1,4);         /* SAS function */
run;
```

Suppose that the IN.WEEKLY variable CODE contains the values cary18593 and apex19624. The SAS function SUBSTR operates during DATA step execution and assigns these values to the variable LOCATION: `cary` and `apex`.

Macro Evaluation Functions

Evaluation functions evaluate arithmetic and logical expressions. They temporarily convert the operands in the argument to numeric values. Then, they perform the operation specified by the operand and convert the result to a character value. The macro processor uses evaluation functions to do the following:

- make character comparisons
- evaluate logical (Boolean) expressions
- assign numeric properties to a token, such as an integer in the argument of a function

For more information, see Chapter 6, “Macro Expressions,” on page 85. The following table lists the macro evaluation functions.

Table 12.5 Macro Evaluation Functions

Function	Description
%EVAL	Evaluates arithmetic and logical expressions using integer arithmetic.

Function	Description
%SYSEVALF	Evaluates arithmetic and logical expressions using floating-point arithmetic.

%EVAL is called automatically by the macro processor to evaluate expressions in the arguments to the statements that perform evaluation in the following functions:

- %QSCAN(*argument*, *n*<, *delimiters*>)
- %QSUBSTR(*argument*, *position*<, *length*>)
- %SCAN(*argument*, *n*<, *delimiters*>)
- %SUBSTR(*argument*, *position*<, *length*>)

Macro Quoting Functions

Macro quoting functions mask special characters and mnemonic operators so that the macro processor interprets them as text instead of elements of the macro language.

The following table lists the macro quoting functions, and also describes the special characters that they mask and when they operate. (Although %QSCAN, %QSUBSTR, and %QUPCASE mask special characters and mnemonic operations in their results, they are not considered quoting functions. Their purpose is to process a character value and not simply to quote a value.) For more information, see “Macro Quoting” on page 96.

Table 12.6 *Macro Quoting Functions*

Function	Description
%BQUOTE, %NRBQUOTE	Mask special characters and mnemonic operators in a resolved value at macro execution. %BQUOTE and %NRBQUOTE are the most powerful functions for masking values at execution time because they do not require that unmatched quotation marks (“ ”) and parentheses (()) be marked.
%QUOTE, %NRQUOTE	Mask special characters and mnemonic operators in a resolved value at macro execution. Unmatched quotation

Function	Description
	marks (“ ”) and parentheses (()) must be marked with a preceding %.
%STR, %NRSTR	Mask special characters and mnemonic operators in constant text at macro compilation. Unmatched quotation marks (“ ”) and parentheses (()) must be marked with a preceding %.
%SUPERQ	Masks all special characters and mnemonic operators at macro execution but prevents resolution of the value.
%UNQUOTE	Unmasks all special characters and mnemonic operators for a value.

Compilation Quoting Functions

%STR and %NRSTR mask special characters and mnemonic operators in values during compilation of a macro definition or a macro language statement in open code. For example, the %STR function prevents the following %LET statement from ending prematurely. It keeps the semicolon in the PROC PRINT statement from being interpreted as the semicolon for the %LET statement.

```
%let printit=%str(proc print; run;);
```

Execution of Macro Quoting Functions

%BQUOTE, %NRBQUOTE, %QUOTE, %NRQUOTE, and %SUPERQ mask special characters and mnemonic operators in values during execution of a macro or a macro language statement in open code. Except for %SUPERQ, these functions instruct the macro processor to resolve a macro expression as far as possible and mask the result. The other quoting functions issue warning messages for any macro variable references or macro invocations that they cannot resolve. %SUPERQ protects the value of a macro variable from any attempt at further resolution.

Of the quoting functions that resolve values during execution, %BQUOTE and %NRBQUOTE are the most flexible. For example, the %BQUOTE function prevents the following %IF statement from producing an error if the macro variable STATE resolves to OR (for Oregon). Without %BQUOTE, the macro processor would interpret the abbreviation for Oregon as the logical operator OR.

```
%if %bquote(&state)=nc %then %put North Carolina Dept. of
Revenue;
```

%SUPERQ fetches the value of a macro variable from the macro symbol table and masks it immediately, preventing the macro processor from attempting to resolve any part of the resolved value. For example, %SUPERQ prevents the following %LET statement from producing an error when it resolves to a value with an ampersand, like `Smith&Jones`. Without %SUPERQ, the macro processor would attempt to resolve `&Jones`.

```
%let testvar=%superq(corpname);
     /* No ampersand in argument to %superq. */
```

(%SUPERQ takes as its argument either a macro variable name without an ampersand or a text expression that yields a macro variable name.)

Quotation Marks and Parentheses without a Match

Syntax errors result if the arguments of %STR, %NRSTR, %QUOTE, and %NRQUOTE contain a quotation mark or parenthesis that does not have a match. To prevent these errors, mark these quotation marks and parentheses by preceding them with a percent sign. For example, write the following to store the value `345)` in macro variable B:

```
%let b=%str(345%));
```

If an argument of %STR, %NRSTR, %QUOTE, or %NRQUOTE contains a percent sign that precedes a quotation mark or parenthesis, use two percent signs (%%) to specify that the argument's percent sign does not mark the quotation mark or parenthesis. For example, write the following to store the value `TITLE "20%";` in macro variable P:

```
%let p=%str(TITLE "20%%";);
```

If the argument for one of these functions contains a character string with the comment symbols `/*` and `-->`, use a %STR function with each character. For example, consider these statements:

```
%let instruct=Comments can start with %str(/)%str(*).;
%put &instruct;
```

They write the following line to the SAS log:

```
   Comments can start with /*
```

Note: Unexpected results can occur if the comment symbols are not quoted with a quoting function.

For more information about macro quoting, see "Macro Quoting" on page 96.

Macro Functions for Double-Byte Character Set (DBCS)

Because East Asian languages have thousands of characters, double (two) bytes of information are needed to represent each character. Each East Asian language usually has more than one DBCS encoding system. SAS processes the DBCS encoding information that is unique for the major East Asian languages. The following table defines the macro functions that support DBCS.

Table 12.7 *Macro Functions for DBCS*

Functions	Description
%KCMPRES	Compresses multiple blanks and removes leading and trailing blanks.
%KINDEX	Returns the position of the first character of a string.
%KLEFT and %QKLEFT	Left-aligns an argument by removing leading blanks.
%KLENGTH	Returns the length of a string.
%KSCAN and %QKSCAN	Searches for a word that is specified by its position in a string.
%KSUBSTR and %QKSUBSTR	%KSUBSTR and %QKSUBSTR produces a substring of a character string.
%KUPCASE and %QKUPCASE	Converts values to uppercase.

For more information, see "Dictionary of Macro Functions for NLS" in *SAS National Language Support (NLS): Reference Guide*.

Other Macro Functions

Seven other macro functions do not fit into the earlier categories, but they provide important information. The following table lists these functions.

Table 12.8 Other Macro Functions

Function	Description
%SYMEXIST	Returns an indication as to whether the named macro variable exists.
%SYMGLOBL	Returns an indication as to whether the named macro variable is global in scope.
%SYMLOCAL	Returns an indication as to whether the named macro variable is local in scope.
%SYSFUNC, %QSYSFUNC	Execute SAS language functions or user-written functions within the macro facility.
%SYSGET	Returns the value of the specified host environment variable. For more information, see the SAS Companion for your operating environment.
%SYSPROD	Reports whether a SAS software product is licensed at the site.

The %SYSFUNC and %QSYSFUNC functions enable the following:

- most of the functions from Base SAS software
- a function written with the SAS/TOOLKIT software
- a function created using the FCMP procedure available to the macro facility

Consider the following examples:

-
  ```
  /* in a DATA step or SCL program */
       dsid=open("Sasuser.Houses","i");
  ```

-
  ```
  /* in the macro facility */
       %let dsid = %sysfunc(open(Sasuser.Houses,i));
  ```

For more information, see “%SYSFUNC Macro Function” on page 356.

Automatic Macro Variables

Automatic macro variables are created by the macro processor and they supply a variety of information. They are useful in programs to check the status of a condition before executing code. You reference automatic macro variables such as &SYSLAST or &SYSJOBID the same way you do macro variables that you create.

CAUTION

Do not create macro variable names that begin with SYS. The three-letter prefix SYS is reserved for use by SAS for automatic macro variables. For a complete list of reserved words in the macro language, see Appendix 1, " Reserved Words in the Macro Facility," on page 483.

For example, suppose you want to include the day and date that your current Compute Server session was invoked. Write the FOOTNOTE statement to reference the automatic macro variables SYSDAY and SYSDATE9:

```
footnote "Report for &sysday, &sysdate9";
```

If the current Compute Server session was invoked on June 13, 2007, macro variable resolution causes SAS to see this statement:

```
FOOTNOTE "Report for Friday, 13JUN2007";
```

All automatic variables except for SYSPBUFF are global and are created when you invoke SAS. The following table lists the automatic macro variables and describes their Read and Write status.

Table 12.9 Automatic Macro Variables

Variable	Read and Write Status
SYSADDRBITS	Read-only
SYSCC	Read and write
SYSCHARWIDTH	Read-only
SYSDATASTEPPHASE	Read-only
SYSDATE	Read-only
SYSDATE9	Read-only
SYSDAY	Read-only
SYSDEVIC	Read and write
SYSDMG	Read and write
SYSDSN	Read and write
SYSENCODING	Read-only
SYSENDIAN	Read-only
SYSENV	Read-only
SYSERR	Read-only

Variable	Read and Write Status
SYSERRORTEXT	Read-only
SYSFILRC	Read and write
SYSHOSTINFOLONG	Read-only
SYSHOSTNAME	Read-only
SYSINCLUDEFILEDEVICE	Read-only
SYSINCLUDEFILEDIR	Read-only
SYSINCLUDEFILEFILEREF	Read-only
SYSINCLUDEFILENAME	Read-only
SYSINDEX	Read-only
SYSINFO	Read-only
SYSJOBID	Read-only
SYSLAST	Read and write
SYSLCKRC	Read and write
SYSLIBRC	Read and write
SYSLOGAPPLNAME	Read-only
SYSMACRONAME	Read-only
SYSMAXLONG	Read-only
SYSMENV	Read-only
SYSNCPU	Read-only
SYSNOBS	Read-only
SYSODSESCAPECHAR	Read-only
SYSODSPATH	Read-only
SYSPARM	Read and write

Variable	Read and Write Status
SYSPBUFF	Read and write
SYSPRINTTOLIST	Read-only
SYSPRINTTOLOG	Read-only
SYSPROCESSID	Read-only
SYSPROCESSMODE	Read-only
SYSPROCESSNAME	Read-only
SYSPROCNAME	Read-only
SYSRC	Read and write
SYSSCP	Read-only
SYSSCPL	Read-only
SYSSITE	Read-only
SYSSIZEOFLONG	Read-only
SYSSIZEOFPTR	Read-only
SYSSIZEOFUNICODE	Read-only
SYSSTARTID	Read-only
SYSSTARTNAME	Read-only
SYSTCPIPHOSTNAME	Read-only
SYSTIME	Read-only
SYSTIMEZONE	Read-only
SYSTIMEZONEIDENT	Read-only
SYSTIMEZONEOFFSET	Read-only
SYSUSERID	Read-only
SYSVER	Read-only

Variable	Read and Write Status
SYSVIYARELEASE	Read-only
SYSVIYAVERSION	Read-only
SYSVLONG	Read-only
SYSVLONG4	Read-only
SYSWARNINGTEXT	Read-only

Interfaces with the Macro Facility

The DATA step, the SAS Component Language, and the SQL procedure provide interfaces with the macro facility. The following tables list the elements that interact with the SAS macro facility.

The DATA step provides elements that enable a program to interact with the macro facility during DATA step execution.

Table 12.10 *Interfaces to the DATA Steps*

Element	Description
EXECUTE routine	Resolves an argument and executes the resolved value at the next step boundary.
RESOLVE function	Resolves the value of a text expression during DATA step execution.
SYMDEL routine	Deletes the indicated macro variable named in the argument.
SYMEXIST function	Returns an indication as to whether the named macro variable exists.
SYMGET function	Returns the value of a macro variable to the DATA step during DATA step execution.
SYMGLOBL function	Returns an indication as to whether the named macro variable is global in scope.

Element	Description
SYMLOCAL function	Returns an indication as to whether the named macro variable is local in scope.
SYMPUT and SYMPUTX routines	Assigns a value produced in a DATA step to a macro variable.

The SAS Component Language (SCL) provides two elements for using the SAS macro facility to define macros and macro variables for SCL programs.

Table 12.11 *Interfaces to the SAS Component Language*

Element	Description
SYMGETN	Returns the value of a global macro variable as a numeric value.
SYMPUTN	Assigns a numeric value to a global macro variable.

The SQL procedure provides a feature for creating and updating macro variables with values produced by the SQL procedure.

Table 12.12 *Interfaces to the SQL Procedure*

Element	Description
INTO	Assigns the result of a calculation or the value of a data column.

For more information, see “Interfaces with the Macro Facility” on page 121.

Selected Autocall Macros Provided with SAS Software

Overview of Provided Autocall Macros

SAS supplies libraries of autocall macros to each SAS site. The libraries that you receive depend on the SAS products licensed at your site. You can use autocall macros without having to define or include them in your programs.

When SAS is installed, the autocall libraries are included in the value of the SASAUTOS system option in the system configuration file. The autocall macros are stored as individual members, each containing a macro definition. Each member has the same name as the macro definition that it contains.

Although the macros available in the autocall libraries supplied by SAS are working utility programs, you can also use them as models for your own routines. In addition, you can call them in macros that you write yourself.

To explore these macro definitions, browse the commented section at the beginning of each member. See the setting of SAS system option SASAUTOS, to find the location of the autocall libraries. To view the SASAUTOS value, use one of the following:

- the OPTIONS command in the SAS windowing environment to open the OPTIONS window
- the OPTIONS procedure
- the VERBOSE system option
- the OPLIST system option

For more information about these options, see “SAS System Options,” in *SAS System Options: Reference*.

The following table lists selected autocall macros.

Table 12.13 Selected Autocall Macros

Macro	Description
CMPRES and QCMPRES	Compresses multiple blanks and removes leading and trailing blanks. QCMPRES masks the result so that special characters and mnemonic operators are treated as text instead of being interpreted by the macro facility.
COMPSTOR	Compiles macros and stores them in a catalog in a permanent SAS library.
DATATYP	Returns the data type of a value.
LEFT and QLEFT	Left-aligns an argument by removing leading blanks. QLEFT masks the result so that special characters and mnemonic operators are treated as text instead of being interpreted by the macro facility.
SYSRC	Returns a value corresponding to an error condition.
TRIM and QTRIM	Trims trailing blanks. QTRIM masks the result so that special characters and mnemonic operators are treated as text instead of being interpreted by the macro facility.
VERIFY	Returns the position of the first character unique to an expression.

Required System Options for Autocall Macros

To use autocall macros, you must set two SAS system options:

MAUTOSOURCE
: activates the autocall facility. NOMAUTOSOURCE disables the autocall facility.

SASAUTOS=*library-specification* | (*library-specification-1*..., *library-specification-n*)
: specifies the autocall library or libraries. For more information, see the SAS companion for your operating system.

If your site has installed the autocall libraries supplied by SAS and uses the standard configuration of SAS software supplied by SAS, you need only to ensure that the SAS system option MAUTOSOURCE is in effect to begin using the autocall macros.

Although the MAUTOLOCDISPLAY system option is not required, it displays the source location of the autocall macros in the SAS log when the autocall macro is invoked. For more information, see "MAUTOLOCDISPLAY System Option" on page 440.

Using Autocall Macros

To use an autocall macro, call it in your program with the statement %*macro-name*. The macro processor searches first in the Work library for a compiled macro definition with that name. If the macro processor does not find a compiled macro and if the MAUTOSOURCE is in effect, the macro processor searches the libraries specified by the SASAUTOS option for a member with that name. When the macro processor finds the member, it does the following:

1. compiles all of the source statements in that member, including all macro definitions
2. executes any open code (macro statements or SAS source statements not within any macro definition) in that member
3. executes the macro with the name that you invoked

After the macro is compiled, it is stored in the Work.SASMacr or Work.SASMacr*n* catalog and is available for use in the Compute Server without having to be recompiled.

Note: The Work.Sasmacr catalog is used to store compiled macros for the primary SAS session. In applications or programs that use side sessions, the catalog used to store compiled macros for each side session is Work.Sasmac*n*, where *n* is a unique integer.

You can also create your own autocall macros and store them in libraries for easy execution. For more information, see Chapter 9, “Storing and Reusing Macros,” on page 137.

Autocall Macros for Double-Byte Character Set (DBCS)

Because East Asian languages have thousands of characters, double (two) bytes of information are needed to represent each character. Each East Asian language usually has more than one DBCS encoding system. SAS processes the DBCS encoding information that is unique for the major East Asian languages. The following table contains definitions for the autocall macros that support DBCS.

Table 12.14 Autocall Macros for DBCS

Autocall Macros	Description
%KLOWCASE and %QKLOWCAS	Changes the uppercase characters to lowercase.
%KTRIM and %QKTRIM	Trims the trailing blanks.
%KVERIFY	Returns the position of the first character unique to an expression.

For more information, see “Dictionary of Autocall Macros for NLS” in *SAS National Language Support (NLS): Reference Guide*.

Selected System Options Used in the Macro Facility

The following table lists the SAS system options that apply to the macro facility.

Table 12.15 System Options Used in the Macro Facility

Option	Description
CMDMAC	Controls command-style macro invocation.
IMPLMAC	Controls statement-style macro invocation.

Option	Description
MACRO	Controls whether the SAS macro language is available.
MAUTOCOMPLOC	Displays in the SAS log the source location of the autocall macros when the autocall macro is compiled.
MAUTOLOCDISPLAY	Displays the source location of the autocall macros in the SAS log when the autocall macro is invoked.
MAUTOLOCINDES	Specifies whether the macro processor prepends the full pathname of the autocall source file to the description field of the catalog entry of compiled auto call macro definition in the Work.SASMacr or Work.SASMacr*n* catalog. [1]
MAUTOSOURCE	Controls whether the macro autocall feature is available.
MCOMPILE	Allows new definitions of macros.
MCOMPILENOTE	Issues a NOTE to the SAS log upon the completion of the compilation of a macro.
MCOVERAGE	Enables the generation of coverage analysis data.
MCOVERAGELOC	Specifies the location of the coverage analysis data file.
MERROR	Controls whether the macro processor issues a warning message when a macro-like name (*%name*) does not match a compiled macro.
MEXECNOTE	Displays macro execution information in the SAS log at macro invocation.
MEXECSIZE	Specifies the maximum macro size that can be executed in memory.
MFILE	Determines whether MPRINT output is routed to an external file.
MINDELIMITER	Specifies the character to be used as the delimiter for the macro IN operator.
MINOPERATOR	Controls whether the macro processor recognizes the IN (#) logical operator.
MLOGIC	Controls whether macro execution is traced for debugging.

Option	Description
MLOGICNEST	Allows the macro nesting information to be displayed in the MLOGIC output in the SAS log.
MPRINT	Controls whether SAS statements generated by macro execution are traced for debugging.
MPRINTNEST	Allows the macro nesting information to be displayed in the MPRINT output in the SAS log.
MRECALL	Controls whether the macro processor searches the autocall libraries for a member that was not found during an earlier search.
MREPLACE	Enables existing macros to be redefined.
MSTORED	Controls whether stored compiled macros are available.
MSYMTABMAX	Specifies the maximum amount of memory available to the macro variable symbol table or tables.
MVARSIZE	Specifies the maximum size for in-memory macro variable values.
SASAUTOS	Specifies one or more autocall libraries.
SASMSTORE	Specifies the libref of a SAS library containing a catalog of stored compiled SAS macros.
SERROR	Controls whether the macro processor issues a warning message when a macro variable reference does not match a macro variable.
SYMBOLGEN	Controls whether the results of resolving macro variable references are displayed for debugging.
SYSPARM	Specifies a character string that can be passed to SAS programs.

1 The Work.Sasmacr catalog is used to store compiled macros for the primary SAS session. In applications or programs that use side sessions, the catalog used to store compiled macros for each side session is Work.Sasmac*n*, where *n* is a unique integer.

PART 2

Macro Language Dictionary

13 AutoCall Macros

AutoCall Macros

SAS supplies libraries of autocall macros to each SAS site. The libraries that you receive depend on the SAS products licensed at your site. You can use autocall macros without having to define or include them in your programs.

Dictionary

%CMPRES Autocall Macro

Compress multiple blanks and remove leading and trailing blanks.

Type:	Autocall macros
Requirement:	MAUTOSOURCE system option

Syntax

%CMPRES (*text* | *text-expression*)

%QCMPRES (*text* | *text-expression*)

Details

Note: Autocall macros are included in a library supplied by SAS Institute. This library might not be installed at your site or might be a site-specific version. If you cannot access this macro or if you want to find out if it is a site-specific version, see your on-site SAS support personnel. For more information, see "Storing and Reusing Macros" on page 137.

The CMPRES and QCMPRES macros compress multiple blanks and remove leading and trailing blanks. If the argument might contain a special character or mnemonic operator, listed below, use %QCMPRES.

CMPRES returns an unquoted result, even if the argument is quoted. QCMPRES produces a result with the following special characters and mnemonic operators masked, so the macro processor interprets them as text instead of as elements of the macro language:

```
& % ' " ( ) + − * / < > = ¬ ^ ~ ; , # blank
AND OR NOT EQ NE LE LT GE GT IN
```

Examples

Example 1: Removing Unnecessary Blanks with %CMPRES

```
%macro createft;
   %let note=The result of &x &op &y is %eval(&x &op &y).;
   %put NOTE: &note;
   %put NOTE: %cmpres(&note);
%mend createft;

data _null_;
   x=5;
   y=10;
   call symput('x',x);    /* Uses BEST12. format */
   call symput('y',y);    /* Uses BEST12. format */
   call symput('op','+'); /* Uses $1. format     */
run;

%createft
```

The CREATEFT macro writes two notes to the SAS log.

```
NOTE: The result of            5 +           10 is 15.
NOTE: The result of 5 + 10 is 15.
```

Example 2: Contrasting %QCMPRES and %CMPRES

```
%let x=5;
%let y=10;
%let a=%nrstr(%eval(&x   +   &y));
%put QCMPRES: %qcmpres(&a);
%put CMPRES: %cmpres(&a);
```

The %PUT statements write the following lines to the log:

```
QCMPRES: %eval(&x + &y)
CMPRES: 15
```

%COMPSTOR Autocall Macro

Compiles macros and stores them in a catalog in a permanent SAS library.

Type:	Autocall macro
Requirement:	MAUTOSOURCE system option

Syntax

%COMPSTOR(PATHNAME=*SAS-data-library*)

Required Argument

SAS-data-library
is the physical name of a SASdata library on your host system. The COMPSTOR macro uses this value to automatically assign a libref. Do not enclose *SAS-data-library* in quotation marks.

Details

Note: Autocall macros are included in a library supplied by SAS. This library might not be installed at your site or might be a site-specific version. If you cannot access this macro or if you want to find out if it is a site-specific version, see your on-site SAS support personnel. For more information, see "Storing and Reusing Macros" on page 137.

The COMPSTOR macro compiles the following autocall macros in a SAS catalog named SASMacr in a permanent SAS library. The overhead of compiling is saved when these macros are called for the first time in a Compute Server session. You can use the COMPSTOR macro as an example of how to create compiled stored macros. For more information about the autocall macros that are supplied by SAS or about using stored compiled macros, see "Storing and Reusing Macros" on page 137.

%CMPRES	%QLEFT
%DATATYP	%QTRIM
%LEFT	%TRIM
%QCMPRES	%VERIFY

%DATATYP Autocall Macro

Returns the data type of a value.

Type:	Autocall macro
Restriction:	Autocall macros are included in a library supplied by SAS. This library might not be installed at your site or might be a site-specific version. If you cannot access this macro or if you want to find out if it is a site-specific version, see your on-site SAS support personnel.
Requirement:	MAUTOSOURCE system option

Syntax

%DATATYP (*text* | *text-expression*)

Details

The DATATYP macro returns a value of `NUMERIC` when an argument consists of digits and a leading plus or minus sign, a decimal, or a scientific or floating-point exponent (E or D in uppercase or lowercase letters). Otherwise, it returns the value `CHAR`.

Note: %DATATYP does not identify hexadecimal numbers.

Example: Determining the Data Type of a Value

```
%macro add(a,b);
%if (%datatyp(&a)=NUMERIC and %datatyp(&b)=NUMERIC) %then %do;
    %put The result is %sysevalf(&a+&b).;
%end;
%else %do;
   %put ERROR:  Addition requires numbers.;
%end;
%mend add;
```

You can invoke the ADD macro:

```
%add(5.1E2,225)
```

The macro then writes this message to the SAS log:

```
The result is 735.
```

Similarly, you can invoke the ADD macro:

```
%add(0c1x, 12)
```

The macro then writes this message to the SAS log:

```
ERROR:  Addition requires numbers.
```

%KVERIFY Autocall Macro

Returns the position of the first character unique to an expression.

Category:	DBCS
Type:	Autocall macro for NLS
Requirement:	MAUTOSOURCE system option

Syntax

%KVERIFY(*source*, *excerpt*)

Required Arguments

source
: text or a text expression. This is the text that you want to examine for characters that do not exist in the excerpt.

excerpt
: text or a text expression. This is the text that defines the set of characters that %KVERIFY uses to examine the source.

Details

Note: Autocall macros are included in a library supplied by SAS. This library might not be installed at your site or might be a site-specific version. If you cannot access this macro or if you want to find out if it is a site-specific version, see your on-site SAS support personnel.

%KVERIFY returns the position of the first character in the source that is not also present in excerpt. If all the characters in source are present in the excerpt, %KVERIFY returns a value of 0.

%LEFT Autocall Macro

Left-align an argument by removing leading blanks.

Type:	Autocall macro
Requirement:	MAUTOSOURCE system option

Syntax

%LEFT(*text* | *text-expression*)

Details

Note: Autocall macros are included in a library supplied by SAS. This library might not be installed at your site or might be a site-specific version. If you cannot access this macro or if you want to find out if it is a site-specific version, see your on-site

SAS support personnel. For more information, see “Storing and Reusing Macros” on page 137.

The LEFT macro and the QLEFT macro both left-align arguments by removing leading blanks. If the argument might contain a special character or mnemonic operator, listed below, use %QLEFT.

%LEFT returns an unquoted result, even if the argument is quoted. %QLEFT produces a result with the following special characters and mnemonic operators masked so that the macro processor interprets them as text instead of as elements of the macro language:

```
& % ' " ( ) + − * / < > = ¬ ^ ~ ; , # blank
AND OR NOT EQ NE LE LT GE GT IN
```

Example: Contrasting %LEFT and %QLEFT

In this example, both the LEFT and QLEFT macros remove leading blanks. However, the QLEFT macro protects the leading & in the macro variable SYSDAY so that it does not resolve.

```
%let d=%nrstr(   &sysday   );
%put *&d* *%qleft(&d)* *%left(&d)*;
```

The %PUT statement writes the following line to the SAS log:

```
*   &sysday    * *&sysday    * *Tuesday   *
```

%LOWCASE Autocall Macro

Change uppercase characters to lowercase.

Type:	Autocall macros
Requirement:	MAUTOSOURCE system option

Syntax

%LOWCASE *text* | *text-expression*()

Without Arguments

See “KLOWCASE Function” in *SAS National Language Support (NLS): Reference Guide*

Details

Note: Autocall macros are included in a library supplied by SAS. This library might not be installed at your site or might be a site-specific version. If you cannot access this macro or if you want to find out if it is a site-specific version, see your on-site SAS support personnel. For more information, see "Storing and Reusing Macros" on page 137.

The %LOWCASE and %QLOWCASE macros change uppercase alphabetic characters to their lowercase equivalents. If the argument might contain a special character or mnemonic operator, listed below, use %QLOWCASE.

%LOWCASE returns a result without quotation marks, even if the argument has quotation marks. %QLOWCASE produces a result with the following special characters and mnemonic operators masked so that the macro processor interprets them as text instead of as elements of the macro language:

```
& % ' " ( ) + − * / < > = ¬ ^ ~ ; , # blank
AND OR NOT EQ NE LE LT GE GT IN
```

Example: Creating a String with Initial Letters Capitalized

```
%macro initcaps(str);
   %global newstr;
   %let newstr=;
   %let lastchar=;
   %do i=1 %to %length(&str);
      %let char=%qsubstr(&str,&i,1);
      %if (&lastchar=%str( ) or &i=1) %then %let char=%qupcase(&char);
      %else %let char=%qlowcase(&char);
      %let newstr=&newstr&char;
      %let lastchar=&char;
   %end;
   &newstr
%mend;

%put %initcaps(%str(sales: COMMAND REFERENCE, VERSION 2, SECOND
EDITION));
```

Submitting this example writes the following to the SAS log:

```
Sales: Command Reference, Version 2, Second Edition
```

%QCMPRES Autocall Macro

Compresses multiple blanks, removes leading and trailing blanks, and returns a result that masks special characters and mnemonic operators.

Type:	Autocall macro
Requirement:	MAUTOSOURCE system option

Syntax

%QCMPRES (*text* | *text-expression*)

Without Arguments

See "%CMPRES Autocall Macro" on page 214.

Details

Note: Autocall macros are included in a library supplied by SAS. This library might not be installed at your site or might be a site-specific version. If you cannot access this macro or if you want to find out if it is a site-specific version, see your on-site SAS support personnel. For more information, see "Storing and Reusing Macros" on page 137.

%QLEFT Autocall Macro

Left-aligns an argument by removing leading blanks and returns a result that masks special characters and mnemonic operators.

Type:	Autocall macro
Requirement:	MAUTOSOURCE system option

Syntax

%QLEFT *text* | *text-expression*()

Without Arguments

See "%LEFT Autocall Macro" on page 218.

Details

Note: Autocall macros are included in a library supplied by SAS. This library might not be installed at your site or might be a site-specific version. If you cannot access this macro or if you want to find out if it is a site-specific version, see your on-site SAS support personnel. For more information, see "Storing and Reusing Macros" on page 137.

The LEFT macro and the QLEFT macro both left-align arguments by removing leading blanks. If the argument might contain a special character or mnemonic operator, listed below, use %QLEFT.

%LEFT returns an unquoted result, even if the argument is quoted. %QLEFT produces a result with the following special characters and mnemonic operators masked so that the macro processor interprets them as text instead of as elements of the macro language:

```
& % ' " ( ) + - * / < > = ¬ ^ ~ ; , # blank
AND OR NOT EQ NE LE LT GE GT IN
```

Example: Contrasting %LEFT and %QLEFT

In this example, both the LEFT and QLEFT macros remove leading blanks. However, the QLEFT macro protects the leading `&` in the macro variable SYSDAY so that it does not resolve.

```
   %let d=%nrstr(    &sysday    );
   %put *&d* *%qleft(&d)* *%left(&d)*;
```

The %PUT statement writes the following line to the SAS log:

```
*    &sysday    * *&sysday    * *Tuesday    *
```

%QLOWCASE Autocall Macro

Changes uppercase characters to lowercase and returns a result that masks special characters and mnemonic operators.

Type:	Autocall macro
Requirement:	MAUTOSOURCE system option

Syntax

%QLOWCASE(*text* | *text-expression*)

Without Arguments

See "%LOWCASE Autocall Macro" on page 219.

See "KLOWCASE Function" in *SAS National Language Support (NLS): Reference Guide*

Details

Note: Autocall macros are included in a library supplied by SAS. This library might not be installed at your site or might be a site-specific version. If you cannot access this macro or if you want to find out if it is a site-specific version, see your on-site SAS support personnel. For more information, see "Storing and Reusing Macros" on page 137.

%QTRIM Autocall Macro

Trims trailing blanks and returns a result that masks special characters and mnemonic operators.

Type:	Autocall macro
Requirement:	MAUTOSOURCE system option

Syntax

%QTRIM (*text* | *text-expression*)

Without Arguments

See "%TRIM Autocall Macro" on page 230.

Details

Note: Autocall macros are included in a library supplied by SAS. This library might not be installed at your site or might be a site-specific version. If you cannot access this macro or if you want to find out if it is a site-specific version, see your on-site SAS support personnel. For more information, see "Storing and Reusing Macros" on page 137.

%SYSRC Autocall Macro

Returns a value corresponding to an error condition.

Type:	Autocall macro
Requirement:	MAUTOSOURCE system option

Syntax

%SYSRC(*character-string*)

Required Argument

character-string
is one of the mnemonic values listed in Table 13.38 on page 225 or a text expression that produces the mnemonic value.

Details

Note: Autocall macros are included in a library supplied by SAS. This library might not be installed at your site or might be a site-specific version. If you cannot access this macro or if you want to find out if it is a site-specific version, see your on-site SAS support personnel. For more information, see "Storing and Reusing Macros" on page 137.

The SYSRC macro enables you to test for return codes produced by SCL functions, the MODIFY statement, and the SET statement with the KEY= option. The SYSRC autocall macro tests for the error conditions by using mnemonic strings rather than the numeric values associated with the error conditions.

When you invoke the SYSRC macro with a mnemonic string, the macro generates a SAS return code. The mnemonics are easier to read than the numeric values, which are not intuitive and subject to change.

You can test for specific errors in SCL functions by comparing the value returned by the function with the value returned by the SYSRC macro with the corresponding mnemonic. To test for errors in the most recent MODIFY or SET statement with the KEY= option, compare the value of the _IORC_ automatic variable with the value returned by the SYSRC macro when you invoke it with the value of the appropriate mnemonic.

The following table lists the mnemonic values to specify with the SYSRC function and a description of the corresponding error.

Table 13.1 Mnemonics for Warning and Error Conditions

Mnemonic	Description
Library Assign or Deassign Messages	
_SEDUPLB	The libref refers to the same physical library as another libref.
_SEIBASN	The specified libref is not assigned.
_SEINUSE	The library or member is not available for use.
_SEINVLB	The library is not in a valid format for the access method.
_SEINVLN	The libref is not valid.
_SELBACC	The action requested cannot be performed because you do not have the required access level on the library.
_SELBUSE	The library is still in use.
_SELGASN	The specified libref is not assigned.
_SENOASN	The libref is not assigned.
_SENOLNM	The libref is not available for use.
_SESEQLB	The library is in sequential (tape) format.
_SWDUPLB	The libref refers to the same physical file as another libref.
_SWNOLIB	The library does not exist.
Fileref Messages	
_SELOGNM	The fileref is assigned to an invalid file.
_SWLNASN	The fileref is not assigned.
SAS Data Set Messages	
_DSENMR	The TRANSACTION data set observation does not exist in the MASTER data set.

Mnemonic	Description
_DSEMTR	Multiple TRANSACTION data set observations do not exist in MASTER data set.
_DSENOM	No matching observation was found in MASTER data set.
_SEBAUTH	The data set has passwords.
_SEBDIND	The index name is not a valid SAS name.
_SEDSMOD	The data set is not open in the correct mode for the specified operation.
_SEDTLEN	The data length is invalid.
_SEINDCF	The new name conflicts with an index name.
_SEINVMD	The open mode is invalid.
_SEINVPN	The physical name is invalid.
_SEMBACC	You do not have the level of access required to open the data set in the requested mode.
_SENOLCK	A record-level lock is not available.
_SENOMAC	Member-level access to the data set is denied.
_SENOSAS	The file is not a SAS data set.
_SEVARCF	The new name conflicts with an existing variable name.
_SWBOF	You tried to read the previous observation when you were on the first observation.
_SWNOWHR	The record no longer satisfies the WHERE clause.
_SWSEQ	The task requires reading observations in a random order, but the engine that you are using allows only sequential access.
_SWWAUG	The WHERE clause has been augmented.
_SWWCLR	The WHERE clause has been cleared.
_SWWREP	The WHERE clause has been replaced.

Mnemonic	Description
SAS File Open and Update Messages	
_SEBDSNM	The filename is not a valid SAS name.
_SEDLREC	The record has been deleted from the file.
_SEFOPEN	The file is currently open.
_SEINVON	The option name is invalid.
_SEINVOV	The option value is invalid.
_SEINVPS	The value of the File Data Buffer pointer is invalid.
_SELOCK	The file is locked by another user.
_SENOACC	You do not have the level of access required to open the file in the requested mode.
_SENOALL	_ALL_ is not allowed as part of a filename in this release.
_SENOCHN	The record was not changed because it would cause a duplicate value for an index that does not allow duplicates.
_SENODEL	Records cannot be deleted from this file.
_SENODLT	The file could not be deleted.
_SENOERT	The file is not open for writing.
_SENOOAC	You are not authorized for the requested open mode.
_SENOOPN	The file or directory is not open.
_SENOPF	The physical file does not exist.
_SENORD	The file is not opened for reading.
_SENORDX	The file is not radix addressable.
_SENOTRD	No record has been read from the file yet.
_SENOUPD	The file cannot be opened for update because the engine is read only.

Mnemonic	Description
_SENOWRT	You do not have Write access to the member.
_SEOBJLK	The file or directory is in exclusive use by another user.
_SERECRD	No records have been read from the input file.
_SWACMEM	Access to the directory will be provided one member at a time.
_SWDLREC	The record has been deleted from file.
_SWEOF	End of file.
_SWNOFLE	The file does not exist.
_SWNOPF	The file or directory does not exist.
_SWNOREP	The file was not replaced because of the NOREPLACE option.
_SWNOTFL	The item pointed to exists but is not a file.
_SWNOUPD	This record cannot be updated at this time.
Library/Member/Entry Messages	
_SEBDMT	The member type specification is invalid.
_SEDLT	The member was not deleted.
_SELKUSR	The library or library member is locked by another user.
_SEMLEN	The member name is too long for this system.
_SENOLKH	The library or library member is not currently locked.
_SENOMEM	The member does not exist.
_SWKNXL	You have locked a library, member, or entry, that does not exist yet.
_SWLKUSR	The library or library member is locked by another user.
_SWLKYOU	You have already locked the library or library member.
_SWNOLKH	The library or library member is not currently locked.

Mnemonic	Description
Miscellaneous Operations	
_SEDEVOF	The device is offline or unavailable.
_SEDSKFL	The disk or tape is full.
_SEINVDV	The device type is invalid.
_SENORNG	There is no write ring in the tape opened for Write access.
_SOK	The function was successful.
_SWINVCC	The carriage-control character is invalid.
_SWNODSK	The device is not a disk.
_SWPAUAC	Pause in I/O, process accumulated data up to this point.
_SWPAUSL	Pause in I/O, slide data window forward and process accumulated data up to this point.
_SWPAUU1	Pause in I/O, extra user control point 1.
_SWPAUU2	Pause in I/O, extra user control point 2.

Comparisons

The SYSRC autocall macro and the SYSRC automatic macro variable are not the same. For more information, see “SYSRC Automatic Macro Variable” on page 269.

Example: Examining the Value of _IORC_

The third DATA step in the follow code illustrates using the autocall macro SYSRC and the automatic variable _IORC_ to control writing a message to the SAS log:

```
data big;
   input id class $;
datalines;
1 A
2 B
3 C
```

```
;

data trans;
   input id class $;
datalines;
1 A
2 B
4 C
;

data big;
   modify big trans;
   by id;
   if _iorc_=%sysrc(_dsenmr) then put 'WARNING: Check ID=' id;
run;
```

In this case, data set TRANS contains an observation that is not in data set BIG, which causes a _DSENMR error. As a result, the following warning is written to the SAS log:

```
WARNING: Check ID=4
```

%TRIM Autocall Macro

Trim trailing blanks.

Type:	Autocall macro
Requirement:	MAUTOSOURCE system option

Syntax

%TRIM(*text* | *text-expression*)

Details

Note: Autocall macros are included in a library supplied by SAS. This library might not be installed at your site or might be a site-specific version. If you cannot access this macro or if you want to find out if it is a site-specific version, see your on-site SAS support personnel. For more information, see "Storing and Reusing Macros" on page 137.

The TRIM macro and the QTRIM macro both trim trailing blanks. If the argument contains a special character or mnemonic operator, listed below, use %QTRIM.

QTRIM produces a result with the following special characters and mnemonic operators masked so that the macro processor interprets them as text instead of as elements of the macro language:

```
& % ' " ( ) + - * / < > = ¬ ° ~ ; , #  blank
AND OR NOT EQ NE LE LT GE GT IN
```

Examples

Example 1: Removing Trailing Blanks

In this example, the TRIM autocall macro removes the trailing blanks from a message that is written to the SAS log.

```
%macro numobs(dsn);
   %local num;
   data _null_;
      set &dsn nobs=count;
      call symput('num', left(put(count,8.)));
      stop;
   run;
   %if &num eq 0 %then
      %put There were NO observations in %upcase(&dsn).;
   %else
      %put There were %trim(&num) observations in data set
%upcase(&dsn).;
%mend numobs;

%numobs(sashelp.class)
```

Invoking the NUMOBS macro generates the following statements:

```
DATA _NULL_;
SET SAMPLE NOBS=COUNT;
CALL SYMPUT('num', LEFT(PUT(COUNT,8.)));
STOP;
RUN;
```

If macro %NUMOBS is run on data set SASHELP.CLASS, the following is written to the SAS log:

```
There were 19 observations in data set SASHELP.CLASS.
```

Example 2: Contrasting %TRIM and %QTRIM

These statements are executed January 28, 1999:

```
%let date=%nrstr(   &sysdate   );
%put *&date* *%qtrim(&date)* *%trim(&date)*;
```

The %PUT statement writes this line to the SAS log:

```
*   &sysdate   * *   &sysdate* *   28JAN99*
```

%VERIFY Autocall Macro

Returns the position of the first character unique to an expression.

Type:	Autocall macro
Requirement:	MAUTOSOURCE system option

Syntax

%VERIFY(*source*, *excerpt*)

Required Arguments

source
is text or a text expression that you want to examine for characters that do not exist in *excerpt*.

excerpt
is text or a text expression. This is the text that defines the set of characters that %VERIFY uses to examine *source*.

Details

Note: Autocall macros are included in a library supplied by SAS. This library might not be installed at your site or might be a site-specific version. If you cannot access this macro or if you want to find out if it is a site-specific version, see your on-site SAS support personnel. For more information, see "Storing and Reusing Macros" on page 137.

%VERIFY returns the position of the first character in *source* that is not also present in *excerpt*. If all characters in *source* are present in *excerpt*, %VERIFY returns 0.

Example: Testing for a Valid Fileref

The ISNAME macro checks a string to verify that it is a valid fileref and prints a message in the SAS log that explains why a string is or is not valid.

```
%macro isname(name);
    %let name=%upcase(&name);
    %if %length(&name)>8 %then
       %put &name: The fileref must be 8 characters or less.;
```

```
      %else %do;
         %let first=ABCDEFGHIJKLMNOPQRSTUVWXYZ_;
         %let all=&first.1234567890;
         %let chk_1st=%verify(%substr(&name,1,1),&first);
         %let chk_rest=%verify(&name,&all);
         %if &chk_rest>0 %then
            %put &name: The fileref cannot contain "%substr(&name,
               &chk_rest,1)".;
         %if &chk_1st>0 %then
            %put &name: The first character cannot be "%substr(&name,
               1,1)".;
         %if (&chk_1st or &chk_rest)=0  %then
            %put &name is a valid fileref.;
      %end;
   %mend isname;

  options nosource;
  %isname(file1)
  %isname(1file)
  %isname(filename1)
  %isname(file$)
  options source;
```

When this program executes, the following is written to the SAS log:

```
FILE1 is a valid fileref.
1FILE: The first character cannot be "1".
FILENAME1: The fileref must be 8 characters or less.
FILE$: The fileref cannot contain "$".
```

14 Automatic Macro Variables

Automatic Macro Variables

Automatic macro variables are created by the macro processor and they supply a variety of information. They are useful in programs to check the status of a condition before executing code.

Dictionary

SYSADDRBITS Automatic Macro Variable

Contains the number of bits of an address.

Type:	Automatic macro variable (read only)

Details

The SYSADDRBITS automatic macro variable contains the number of bits needed for an address.

SYSCC Automatic Macro Variable

Contains the current condition code that SAS returns to your operating environment (the operating environment condition code).

Type:	Automatic macro variable (read and write)
Restriction:	The SYSCC macro variable does not capture condition codes from CAS statements.To detect an error from a CAS statement, use the CASSTMTERR macro variable or the SYSERRORTEXT on page 249 macro variable instead.
Notes:	The value of SYSCC might not match the return code that is returned by the operating system.
	When ERRORCHECK=NORMAL, the return code is 0, even if an error exists in a LIBNAME or FILENAME statement, or in a LOCK statement in SAS/SHARE software. Also, the SAS job or session does not end abnormally when the %INCLUDE statement fails due to a nonexistent file. For more information, see "ERRORCHECK= System Option" in *SAS System Options: Reference*.
See:	

Details

SYSCC is a read and write automatic macro variable that enables you to reset the job condition code and to recover from conditions that prevent subsequent steps from running.

Upon exit, SAS translates this condition code to a return code that has a meaningful value for the operating environment. A normal exit internally to SAS is 0. The host code translates the internal value to a meaningful condition code by the host operating environment. &SYSCC of 0 at SAS termination is the value of success for that operating environment's return code.

Note: When the ERRORCHECK= system option is set at NORMAL, the value of SYSCC will be 0 even if an error exists in a LIBNAME or FILENAME statement, or in a LOCK statement in SAS/SHARE software. The value of SYSCC will also be 0 when the %INCLUDE statement fails due to a nonexistent file. For more information, see the "ERRORCHECK= System Option" in *SAS System Options: Reference*.

SYSCHARWIDTH Automatic Macro Variable

Contains the character width value.

Type:	Automatic macro variable (read only)

Details

The character width value is either 1 (narrow) or 2 (wide).

SYSDATASTEPPHASE Automatic Macro Variable

Indicates the current running phase of the DATA step.

Type:	Automatic macro variable (read only)

Details

Macros are designed to only execute during the compilation of a DATA step or during the execution of a DATA step. The new automatic macro variable SYSDATASTEPPHASE ensures that the macro is being executed as part of the proper phase of a DATA step.

The value of SYSDATASTEPPHASE automatic macro variable indicates the current active phase of the DATA step. When the DATA step is not active, the value of SYSDATASTEPPHASE is null. The following are possible values of SYSDATASTEPPHASE automatic macro variable :

- INITIALIZATION

- COMPILATION
- RESOLUTION
- EXECUTION
- AUTO-LOADING STORED PROGRAM
- COMPILATION — STORED PROGRAM LOADING
- LOADING STORED PROGRAM
- AUTO-SAVING STORED PROGRAM
- SAVING STORED PROGRAM

Any non-null value, other than EXECUTION, should be considered as part of the DATA step compilation process.

Examples

Example 1: EXECUTION Phase

```
data null;
   x=1;
   /* Placing the argument in single quote marks delays the   */
   /* evaluation until after the DATA step has been compiled. */
   call execute('%put &sysdatastepphase;');
   put x=;
run;
```

The following is written to the SAS log.

```
EXECUTION
NOTE: DATA statement used (Total process time):
      real time           0.04 seconds
      cpu time            0.01 seconds

x=1
```

Example 2: COMPILATION Phase

```
data null;
   call symput("phase", "&sysdatastepphase");
run;

%put &=phase;
```

The following is written to the SAS log.

```
PHASE=COMPILATION
```

SYSDATE Automatic Macro Variable

Contains the date on which a SAS job or session began executing.

Type:	Automatic macro variable (read only)
See:	"SYSDATE9 Automatic Macro Variable" on page 240

Details

SYSDATE contains a SAS date value in the DATE7. format, which displays a two-digit day, the first three letters of the month name, and a two-digit year. The date does not change during the individual job or session. For example, you could use SYSDATE in programs to check the date before you execute code that you want to run on certain dates of the month.

Example: Formatting a SYSDATE Value

Macro FDATE assigns a format that you specify to the value of SYSDATE:

```
%macro fdate(fmt);
   %global fdate;
   data _null_;
      call symput("fdate",left(put("&sysdate"d,&fmt)));
   run;
%mend fdate;

%fdate(worddate.)
```

If you execute this macro on September 1, 2023, for example, SAS sees the statements:

```
DATA _NULL_;
   CALL SYMPUT("FDATE",LEFT(PUT("01SEP23"D,WORDDATE.)));
RUN;
%PUT "Tests for September 1, 2023";
```

For another method of formatting the current date, see the %SYSFUNC and %QSYSFUNC functions.

SYSDATE9 Automatic Macro Variable

Contains the date on which a SAS job or session began executing.

Type:	Automatic macro variable (read only)

See: "SYSDATE Automatic Macro Variable" on page 240

Details

SYSDATE9 contains a SAS date value in the DATE9. format, which displays a two-digit day, the first three letters of the month name, and a four-digit year. The date does not change during the individual job or session. For example, you could use SYSDATE9 in programs to check the date before you execute code that you want to run on certain dates of the month.

Example: Formatting a SYSDATE9 Value

Macro FDATE assigns a format that you specify to the value of SYSDATE9:

```
%macro fdate(fmt);
     %global fdate;
   data _null_;
      call symput("fdate",left(put("&sysdate9"d,&fmt)));
   run;
%mend fdate;

%fdate(worddate.)

%put Tests for &fdate;
```

If you execute this macro on September 1, 2023, for example, SAS sees the statements:

```
DATA _NULL_;
   CALL SYMPUT("FDATE",LEFT(PUT("01SEP2023"D,WORDDATE.)));
RUN;
%PUT "Tests for September 1, 2023";
```

For another method of formatting the current date, see the %SYSFUNC and %QSYSFUNC functions.

SYSDAY Automatic Macro Variable

Contains the day of the week on which a SAS job or session began executing.

Type: Automatic macro variable (read only)

Details

You can use SYSDAY to check the current day before executing code that you want to run on certain days of the week, provided you initialized your Compute Server session today.

Example: Identifying the Day When a Compute Server Session Started

The following statement identifies the day and date on which a Compute Server session started running.

```
%put This SAS session started running on: &sysday, &sysdate9.;
```

When this statement executes on Wednesday, December 19, 2007 for a Compute Server session that began executing on Monday, December 17, 2007, the following line is written to the SAS log:

```
This SAS session started running on: Monday, 17DEC2007
```

SYSDEVIC Automatic Macro Variable

Contains the name of the current graphics device.

Type: Automatic macro variable (read and write)

See:

Details

The graphics device is used by products that use SAS/GRAPH software. The current graphics device is determined by the DEVICE system option or the DEVICE graphics option. See "DEVICE= System Option" in *SAS/GRAPH: Reference* or "DEVICE= Graphics Option" in *SAS/GRAPH: Reference*. You can specify the graphics device in a configuration file. For information about graphics devices and SAS/GRAPH software, see *SAS/GRAPH: Reference*.

Note: The macro processor always stores the value of SYSDEVIC in unquoted form. To quote the resolved value of SYSDEVIC, use the %SUPERQ macro quoting function.

Comparisons

Assigning a value to SYSDEVIC is the same as specifying a value for the DEVICE= system option.

SYSDMG Automatic Macro Variable

Contains a return code that reflects an action taken on a damaged data set.

Type:	Automatic macro variable (read and write)
Default:	0

Details

You can use the value of SYSDMG as a condition to determine further action to take.

SYSDMG can contain the following values:

Table 14.1 SYSDMG Values and Descriptions

Value	Description
0	No repair of damaged data sets in this session. (Default)
1	One or more automatic repairs of damaged data sets have occurred.
2	One or more user-requested repairs of damaged data sets have occurred.
3	One or more opens failed because the file was damaged.
4	One or more SAS tasks were terminated because of a damaged data set.
5	One or more automatic repairs of damaged data sets have occurred; the last-repaired data set has index file removed, as requested.
6	One or more user requested repairs have occurred; the last-repaired data set has index file removed, as requested.

SYSDSN Automatic Macro Variable

Contains the libref and name of the most recently created SAS data set.

Type:	Automatic macro variable (read and write)
Tip:	Use autocall macro %TRIM to remove trailing blanks.
See:	"SYSLAST Automatic Macro Variable" on page 257

Details

The libref and data set name are displayed in two left-aligned fields. If no SAS data set has been created in the current program, SYSDSN returns 8 blanks followed by _NULL_ followed by 26 blanks.

Note: The macro processor always stores the value of SYSDSN in unquoted form. To quote the resolved value of SYSDSN, use the %SUPERQ macro quoting function.

Comparisons

- Assigning a value to SYSDSN is the same as specifying a value for the _LAST_= system option.
- The value of SYSLAST is often more useful than SYSDSN because the value of SYSLAST is formatted so that you can insert a reference to it directly into SAS code in place of a data set name.

Example: Comparing Values Produced by SYSDSN and SYSLAST

Create a data set named Work.Teams and enter the %PUT statements:

```
data teams;
  input color $15. @16 team_name $15. @32 game1 game2 game3;
datalines;
Green          Crickets        10 7 8
Blue           Sea Otters      10 6 7
Yellow         Stingers        9 10 9
Red            Hot Ants        8 9 9
Purple         Cats            9 9 9
;
```

```
%put Sysdsn produces:  *&sysdsn*;
%put Syslast produces: *&syslast*;
```

The following is written to the SAS log:

```
Sysdsn produces:  *WORK    TEAMS                            *
Syslast produces: *WORK.TEAMS                            *
```

When the libref name contains fewer than 8 characters, SYSDSN uses trailing blanks to pad the libref name to 8 characters. Likewise, when the data set name contains fewer than 32 characters, SYSDSN uses trailing blanks to pad the data set name to 32 characters. SYSDSN does not display a period between the libref and data set name fields. You can use autocall macro %TRIM in the %PUT statements to remove trailing blanks from the data set name, if needed, as follows:

```
%put Sysdsn produces:  *%trim(&sysdsn)*;
%put Syslast produces: *%trim(&syslast)*;
```

The following is written to the SAS log:

```
Sysdsn produces:  *WORK    TEAMS*
Syslast produces: *WORK.TEAMS*
```

SYSENCODING Automatic Macro Variable

Contains the name of the Compute Server session encoding.

Type: Automatic macro variable (read only)

Details

SYSENCODING displays the name with a maximum length of 12 bytes.

Example: Using SYSENCODING to Display the Compute Server Session Encoding

The following statement displays the encoding for the Compute Server session:

```
%put The encoding for this session is: &sysencoding;
```

When this statement executes, the following comment is written to the SAS log:

```
The encoding for this session is: wlatin1
```

SYSENDIAN Automatic Macro Variable

Contains an indication of the byte order of the current session. The possible values are LITTLE or BIG.

Type:	Automatic macro variable (read only)

Details

The SYSENDIAN automatic macro variable indicates the byte order of the current Compute Server session. There are two possible values: LITTLE and BIG.

SYSENV Automatic Macro Variable

Reports whether SAS is running interactively.

Type:	Automatic macro variable (read only)
Default:	BACK
See:	

Details

The value of SYSENV is independent of the source of input. The following are values for SYSENV:

FORE
: when the SAS system option TERMINAL is in effect. For example, the value is FORE when you run SAS interactively.

BACK
: when the SAS system option NOTERMINAL is in effect. For example, the value is BACK when you submit a SAS job in batch mode.

You can use SYSENV to check the execution mode before submitting code that requires interactive processing. For more information, see the SAS documentation for your operating environment.

Operating Environment Information: Some operating environments do not support the submission of jobs in batch mode. In this case the value of SYSENV is always FORE. For more information, see the SAS documentation for your operating environment.

SYSERR Automatic Macro Variable

Contains a return code status set by some SAS procedures and the DATA step.

Type:	Automatic macro variable (read only)

Details

You can use the value of SYSERR as a condition to determine further action to take or to decide which parts of a SAS program to execute. SYSERR is used to detect major system errors, such as out of memory or failure of the component system when used in some procedures and DATA steps. SYSERR automatic macro variable is reset at each step boundary. For the return code of a complete job, see "SYSCC Automatic Macro Variable" on page 237.

SYSERR can contain the following values:

Table 14.2 SYSERR Values

Value	Description
0	Execution completed successfully and without warning messages.
1	Execution was canceled by a user with a RUN CANCEL statement.
2	Execution was canceled by a user with an ATTN or BREAK command.
3	An error in a program run in batch or non-interactive mode caused SAS to enter syntax-check mode.
4	Execution completed successfully but with warning messages.
5	Execution was canceled by a user with an ABORT CANCEL statement.
6	Execution was canceled by a user with an ABORT CANCEL FILE statement.
>6	An error occurred. The value returned is procedure-dependent.

The following table contains warning return codes. The codes do not indicate any specific problems. These codes are guidelines to identify the nature of a problem.

Table 14.3 SYSERR Warning Codes

Warning Code	Description
108	Problem with one or more BY groups
112	Error with one or more BY groups
116	Memory problems with one or more BY groups
120	I/O problems with one or more BY groups

The following table contains error return codes. The codes do not indicate any specific problems. These codes are guidelines to identify the nature of a problem.

Table 14.4 SYSERR Error Codes

Error Code	Description
1008	General data problem
1012	General error condition
1016	Out-of-memory condition
1020	I/O problem
2000	Semantic action problem
2001	Attribute processing problem
3000	Syntax error
4000	Not a valid procedure
9999	Bug in the procedure
20000	A step was stopped or an ABORT statement was issued.
20001	An ABORT RETURN statement was issued.
20002	An ABORT ABEND statement was issued.
25000	Severe system error. The system cannot initialize or continue.

Example: Using SYSERR

The example creates an error message and uses %PUT &SYSERR to write the return code number (1012) to the SAS log.

```
data NULL;
   set doesnotexist;
run;
%put &=syserr;
```

The following SAS log output contains the return code number:

```
SYSERR=1012
```

To retrieve error and warning text instead of the return code number, see "SYSERRORTEXT Automatic Macro Variable" on page 249 and "SYSWARNINGTEXT Automatic Macro Variable" on page 281.

SYSERRORTEXT Automatic Macro Variable

Contains the text of the last error message formatted for display in the SAS log.

Type:	Automatic macro variable (read and write starting with 2024.05, read only in 2024.04 and prior releases)

Details

The value of SYSERRORTEXT is the text of the last error message generated in the SAS log. For a list of SYSERR warnings and errors, see "SYSERR Automatic Macro Variable" on page 247.

Note: If the last error message text that was generated contains an `&` or `%` and you are using the %PUT statement, you must use the %SUPERQ macro quoting function to mask the special characters to prevent further resolution of the value. The following example uses the %PUT statement and the %SUPERQ macro quoting function:

```
%put %superq(syserrortext);
```

For more information, see "%SUPERQ Macro Function" on page 347.

Example: Using SYSERRORTEXT

This example creates an error message:

```
data NULL;
   set doesnotexist;
run;
```

```
%put &syserrortext;
```

When these statements are executed, the following record is written to the SAS log:

```
1   data NULL;
2   set doesnotexist;
ERROR: File WORK.DOESNOTEXIST.DATA does not exist.
3   run;
NOTE: The SAS System stopped processing this step because of errors.
WARNING: The data set WORK.NULL might be incomplete. When this step was
         stopped there were 0 observations and 0 variables.
NOTE: DATA statement used (Total process time):
real time       11.16 seconds
cpu time        0.07 seconds
4  %put &syserrortext;
File WORK.DOESNOTEXIST.DATA does not exist.
```

SYSFILRC Automatic Macro Variable

Contains the return code from the last FILENAME statement.

Type: Automatic macro variable (read and write)

Details

SYSFILRC checks whether the file or storage location referenced by the last FILENAME statement exists. You can use SYSFILRC to confirm that a file or location is allocated before attempting to access an external file.

The following are values for SYSFILRC:

Table 14.5 *SYSFILRC Values and Descriptions*

Value	Description
0	The last FILENAME statement executed correctly.
≠0	The last FILENAME statement did not execute correctly.

SYSHOSTINFOLONG Automatic Macro Variable

Contains the operating environment information that is displayed when the HOSTINFOLONG option is specified.

Type:	Automatic macro variable (read-only)

Details

Contains the operating environment information that is displayed when the HOSTINFOLONG option is specified. For more information, see "HOSTINFOLONG System Option" in *SAS System Options: Reference*.

SYSHOSTNAME Automatic Macro Variable

Contains the host name of the computer that is running the SAS process.

Type:	Automatic macro variable (read only)

Details

SYSHOSTNAME contains the host name of the system that is running a single TCPIP stack. If you are running multiple TCPIP stacks, use the SYSTCPIPHOSTNAME automatic macro variable. For more information about TCPIP stacks, see your SAS host companion documentation.

See Also

"SYSTCPIPHOSTNAME Automatic Macro Variable" on page 273

SYSINCLUDEFILEDEVICE Automatic Macro Variable

Contains the device type of the current %INCLUDE file.

Type:	Automatic macro variable (read-only)
Restriction:	This variable does not resolve in a macro definition that is included with the %INCLUDE statement. To resolve in included code, the reference must be outside of a macro definition. See "Example 2: Using the SYSINCLUDEFILE Automatic Macro Variables in Included Macro Definitions" on page 253.

Details

You can use SYSINCLUDEFILEDEVICE to determine the type of device on which an %INCLUDE file is stored. In included code, use a %PUT statement to write the value of SYSINCLUDEFILEDEVICE to the SAS log. You can use other automatic macro variables to write additional information to the log.

TIP Use %PUT _AUTOMATIC_ to see a list of all available automatic macro variables.

For an example, see "Example 1: Using the SYSINCLUDEFILE Automatic Macro Variables in Included Code" on page 252.

SYSINCLUDEFILEDEVICE does not resolve in a macro that is defined in included code. To reference SYSINCLUDEFILEDEVICE in an included macro definition, first, assign the SYSINCLUDEFILEDEVICE value to a global macro variable. The global macro variable assignment must be outside of the macro definition. Next, use the global macro variable in the macro definition. This is also required for automatic macro variables SYSINCLUDEFILEDIR, SYSINCLUDEFILEFILEREF, and SYSINCLUDEFILENAME. For an example, see "Example 2: Using the SYSINCLUDEFILE Automatic Macro Variables in Included Macro Definitions" on page 253.

Examples

Example 1: Using the SYSINCLUDEFILE Automatic Macro Variables in Included Code

You can use SYSINCLUDEFILEDEVICE with automatic macro variables SYSINCLUDEFILEDIR, SYSINCLUDEFILENAME, and SYSINCLUDEFILEFILEREF to write information about an %INCLUDE file to the SAS log. Here is a simple example that writes to the SAS log information about an include file. This code is stored in file Myincfile.txt in directory **c:\sas\includefiles**:

```
/* Include file myincfile.txt */
options nosource;
%put NOTE: Begin include file &sysincludefilename.:;
%put NOTE- %str(   File path:   )&sysincludefiledir
\&sysincludefilename;
%put NOTE- %str(   Device type: )&sysincludefiledevice;
%put NOTE- %str(   Fileref:     )&sysincludefilefileref;
options source;

%put More code ...;

%put NOTE: End include file &sysincludefilename.;
```

The following statements include the code in file Myincfile.txt in the current SAS session:

```
filename myincfn "c:\sas\includefiles\myincfile.txt";
%include myincfn;
filename myincfn clear;
```

Here is the output.

```
NOTE: Begin include file myincfile.txt:
        File path:   c:\sas\includefiles\myincfile.txt
        Device type: DISK
        Fileref:     MYINCFN
More code ...
NOTE: End include file myincfile.txt
```

Example 2: Using the SYSINCLUDEFILE Automatic Macro Variables in Included Macro Definitions

To use SYSINCLUDEFILEDEVICE, SYSINCLUDEFILEDIR, SYSINCLUDEFILENAME, and SYSINCLUDEFILEFILEREF in a macro definition in included code, assign the automatic macro variables to global macro variables, and then use the global macro variables in the macro definition. Here is a simple example that writes to the SAS log information about a macro that is defined in included code. This code is stored in file Myincfile.txt in directory **`c:\sas\includefiles`**:

```
/* Include file myincfile.txt */
%let incdev=%trim(&sysincludefiledevice);
%let incdir=&sysincludefiledir;
%let incfn=&sysincludefilename;
%let incfref=&sysincludefilefileref;

%macro mymacro;
   options nosource;
   %put NOTE: Begin macro %nrstr(%mymacro).;
   %put NOTE- %str(   Defined in:  )&incdir\&incfn;
   %put NOTE- %str(   Device type: )&incdev;
   %put NOTE- %str(   Fileref:     )&incfref;
   options source;

   %put More macro code ...;

   %put NOTE: End macro %nrstr(%mymacro).;
%mend mymacro;

/* More code ... */

/* Run macro %mymacro */
%mymacro;
```

The following statements include the code in file Myincfile.txt in the current SAS session:

```
filename myincfn "c:\sas\includefiles\myincfile.txt";
%include myincfn;
filename myincfn clear;
```

Here is the output.

```
NOTE: Begin macro %mymacro.
         Defined in:  c:\sas\includefiles\myincfile2.txt
         Device type: DISK
         Fileref:     MYINCFN
More macro code ...
NOTE: End macro %mymacro.
```

SYSINCLUDEFILEDIR Automatic Macro Variable

Contains the directory where the current %INCLUDE file was found.

Type:	Automatic macro variable (read-only)
Restriction:	This variable does not resolve in a macro definition that is included with the %INCLUDE statement. To resolve in included code, the reference must be outside of a macro definition. See "Example 2: Using the SYSINCLUDEFILE Automatic Macro Variables in Included Macro Definitions" on page 253.

Details

You can use SYSINCLUDEFILEDIR to determine the directory in which an %INCLUDE file is stored. In included code, use a %PUT statement to write the value of SYSINCLUDEFILEDIR to the SAS log. You can use other automatic macro variables to write additional information to the log.

TIP Use %PUT _AUTOMATIC_ to see a list of all available automatic macro variables.

For an example, see "Example 1: Using the SYSINCLUDEFILE Automatic Macro Variables in Included Code" on page 252.

SYSINCLUDEFILEDIR does not resolve in a macro that is defined in included code. To reference SYSINCLUDEFILEDIR in an included macro definition, first, assign the SYSINCLUDEFILEDIR value to a global macro variable. The global macro variable assignment must be outside of the macro definition. Next, use the global macro variable in the macro definition. This is also required for automatic macro variables SYSINCLUDEFILEDEVICE, SYSINCLUDEFILEFILEREF, and SYSINCLUDEFILENAME. For an example, see "Example 2: Using the SYSINCLUDEFILE Automatic Macro Variables in Included Macro Definitions" on page 253.

SYSINCLUDEFILEFILEREF Automatic Macro Variable

Contains the fileref that is associated with the current %INCLUDE file or blank.

Type:	Automatic macro variable (read-only)
Restriction:	This variable does not resolve in a macro definition that is included with the %INCLUDE statement. To resolve in included code, the reference must be outside of a macro definition. See “Example 2: Using the SYSINCLUDEFILE Automatic Macro Variables in Included Macro Definitions” on page 253.

Details

You can use SYSINCLUDEFILEFILEREF to determine the name of the fileref that was used to reference an %INCLUDE file. In included code, use a %PUT statement to write the value of SYSINCLUDEFILEFILEREF to the SAS log. You can use other automatic macro variables to write additional information to the log.

TIP Use %PUT _AUTOMATIC_ to see a list of all available automatic macro variables.

For an example, see “Example 1: Using the SYSINCLUDEFILE Automatic Macro Variables in Included Code” on page 252.

SYSINCLUDEFILEFILEREF does not resolve in a macro that is defined in included code. To reference SYSINCLUDEFILEFILEREF in an included macro definition, first, assign the SYSINCLUDEFILEFILEREF value to a global macro variable. The global macro variable assignment must be outside of the macro definition. Next, use the global macro variable in the macro definition. This is also required for automatic macro variables SYSINCLUDEFILEDEVICE, SYSINCLUDEFILEDIR, and SYSINCLUDEFILENAME. For an example, see “Example 2: Using the SYSINCLUDEFILE Automatic Macro Variables in Included Macro Definitions” on page 253.

SYSINCLUDEFILENAME Automatic Macro Variable

Contains the filename of the current %INCLUDE file.

Type:	Automatic macro variable (read-only)
Restriction:	This variable does not resolve in a macro definition that is included with the %INCLUDE statement. To resolve in included code, the reference must be outside

of a macro definition. See “Example 2: Using the SYSINCLUDEFILE Automatic Macro Variables in Included Macro Definitions” on page 253.

Details

You can use SYSINCLUDEFILENAME to determine the name of an %INCLUDE file. In included code, use a %PUT statement to write the value of SYSINCLUDEFILENAME to the SAS log. You can use other automatic macro variables to write additional information to the log.

TIP Use %PUT _AUTOMATIC_ to see a list of all available automatic macro variables.

For an example, see “Example 1: Using the SYSINCLUDEFILE Automatic Macro Variables in Included Code” on page 252.

SYSINCLUDEFILENAME does not resolve in a macro that is defined in included code. To reference SYSINCLUDEFILENAME in an included macro definition, first, assign the SYSINCLUDEFILENAME value to a global macro variable. The global macro variable assignment must be outside of the macro definition. Next, use the global macro variable in the macro definition. This is also required for automatic macro variables SYSINCLUDEFILEDEVICE, SYSINCLUDEFILEDIR, and SYSINCLUDEFILEFILEREF. For an example, see “Example 2: Using the SYSINCLUDEFILE Automatic Macro Variables in Included Macro Definitions” on page 253.

SYSINDEX Automatic Macro Variable

Contains the number of macros that have started execution in the current SAS job or session.

Type:	Automatic macro variable (read only)

Details

You can use SYSINDEX in a program that uses macros when you need a unique number that changes after each macro invocation.

SYSINFO Automatic Macro Variable

Contains return codes provided by some SAS procedures.

Type:	Automatic macro variable (read only)

Details

Values of SYSINFO are described with the procedures that use it. You can use the value of SYSINFO as a condition for determining further action to take or parts of a SAS program to execute.

For example, PROC COMPARE, which compares two data sets, uses SYSINFO to store a value that provides information about the result of the comparison.

SYSJOBID Automatic Macro Variable

Lists the process identification number (PID) of the process that is executing SAS.

Type:	Automatic macro variable (read only)

Details

The value stored in SYSJOBID depends on the operating environment that you use to run SAS. You can use SYSJOBID to check who is currently executing the job to restrict certain processing or to issue commands that are specific to a user.

SYSLAST Automatic Macro Variable

Contains the name of the SAS data file created most recently.

Type:	Automatic macro variable (read and write)
Tip:	Use autocall macro %TRIM to remove trailing blanks.
See:	"SYSDSN Automatic Macro Variable" on page 244

Details

The name is stored in the form *libref.dataset*. You can insert a reference to SYSLAST directly into SAS code in place of a data set name. Use autocall macro %TRIM to remove trailing blanks, if needed. If no SAS data set has been created in the current program, the value of SYSLAST is _NULL_, with no leading or trailing blanks.

Note: The macro processor always stores the value of SYSLAST in unquoted form. To quote the resolved value of SYSLAST, use the %SUPERQ macro quoting function.

Comparisons

- Assigning a value to SYSLAST is the same as specifying a value for the _LAST_= system option.
- The value of SYSLAST is often more useful than SYSDSN because the value of SYSLAST is formatted so that you can insert a reference to it directly into SAS code in place of a data set name.

Example: Comparing Values Produced by SYSLAST and SYSDSN

Create the data set FirstLib.SalesRpt and then enter the following statements:

```
%put Sysdsn produces:  *&sysdsn*;
%put Syslast produces: *&syslast*;
```

When these statements are executed, the following is written to the SAS log:

```
Sysdsn produces:  *FIRSTLIBSALESRPT                        *
Syslast produces: *FIRSTLIB.SALESRPT                        *
```

The name stored in SYSLAST contains the period between the libref and data set name. To remove trailing blanks, use autocall macro %TRIM as follows:

```
%put Sysdsn produces:  *%trim(&sysdsn)*;
%put Syslast produces: *%trim(&syslast)*;
```

The following is written to the SAS log:

```
Sysdsn produces:  *FIRSTLIBSALESRPT*
Syslast produces: *FIRSTLIB.SALESRPT*
```

SYSLCKRC Automatic Macro Variable

Contains the return code from the most recent LOCK statement.

Type:	Automatic macro variable (read and write)

Details

The LOCK statement is a Base SAS software statement used to acquire and release an exclusive lock on data objects in data libraries accessed through SAS/SHARE software. The following are values for SYSLCKRC:

Table 14.6 *LCKRC Values and Descriptions*

Value	Description
0	The last LOCK statement was successful.
>0	The last LOCK statement was not successful.
<0	The last LOCK statement was completed, but a WARNING or NOTE was written to the SAS log.

For more information, see the documentation for SAS/SHARE software.

SYSLIBRC Automatic Macro Variable

Contains the return code from the last LIBNAME statement.

Type:	Automatic macro variable (read and write)

Details

The code reports whether the last LIBNAME statement executed correctly. SYSLIBRC checks whether the SAS library referenced by the last LIBNAME statement exists. For example, you could use SYSLIBRC to confirm that a libref is allocated before you attempt to access a permanent data set.

The following are values for SYSLIBRC:

Table 14.7 *SYSLIBRC Values and Descriptions*

Value	Description
0	The last LIBNAME statement executed correctly.
≠0	The last LIBNAME statement did not execute correctly.

SYSLOGAPPLNAME Automatic Macro Variable

Contains the value of the LOGAPPLNAME= system option.

Type:	Automatic macro variable (read only)

Default:	null

Details

The following code, when submitted from the current Compute Server session, writes the LOGAPPLNAME for the current Compute Server session to the log:

```
%put &syslogapplname;
```

SYSMACRONAME Automatic Macro Variable

Returns the name of the currently executing macro.

Type:	Automatic macro variable (read only)

Details

When referenced outside of an executing macro, SYSMACRONAME returns the null string.

SYSMAXLONG Automatic Macro Variable

Returns the maximum long integer value allowed under Linux, which is 9,007,199,254,740,992.

Type:	Automatic macro variable (read-only)

SYSMENV Automatic Macro Variable

Contains the invocation status of the macro that is currently executing.

Type:	Automatic macro variable (read only)

Details

The following are values for SYSMENV:

Table 14.8 SMENV Values and Descriptions

Value	Description
`S`	The macro currently executing was invoked as part of a SAS program.
`D`	The macro currently executing was invoked from the command line of a SAS window.

SYSNCPU Automatic Macro Variable

Contains the current number of processors available to SAS for computations.

Type:	Automatic Macro Variable (Read Only)

Details

SYSNCPU is an automatic macro variable that provides the current value of the CPUCOUNT option. For more information, see "CPUCOUNT= System Option" in *SAS System Options: Reference*.

Comparisons

The following example shows the option CPUCOUNT set to 265.

```
options cpucount=265;
%put &=sysncpu;
```

Here is the result:

```
   SYSNCPU=265
```

SYSNOBS Automatic Macro Variable

Contains the number of observations that exist in the last data set that was closed by the previous procedure or DATA step.

Type:	Automatic macro variable (read and write)

Details

SYSNOBS automatic macro variable contains the number of observations that exist in the last data set that was closed by the previous procedure or DATA step.

Note: If the number of observations for the data set was not calculated by the previous procedure or DATA step, the value of SYSNOBS is set to -1.

SYSODSESCAPECHAR Automatic Macro Variable

Displays the value of the ODS ESCAPECHAR= from within the program.

Type: Automatic macro variable (read only)

Details

SYSODSESCAPECHAR automatic macro variable contains the hex representation of the current ODS escape character. Here is an example:

```
ods escapechar='1';
%put &=SYSODSescapeChar;

ods escapechar='a';
%put &=SYSODSescapeChar;

ods escapechar='#';
%put &=SYSODSescapeChar;

ods escapechar="03"x;
%put &=SYSODSescapeChar;

ods escapechar='%';
%put &=SYSODSescapeChar;

ods escapechar='abc';
%put &=SYSODSescapeChar;
```

The following is written to the SAS log:

```
1          ods escapechar='1';
2          %put &=SYSODSescapeChar;
SYSODSESCAPECHAR=31
3
4          ods escapechar='a';
5          %put &=SYSODSescapeChar;
SYSODSESCAPECHAR=61
6
7          ods escapechar='#';
8          %put &=SYSODSescapeChar;
SYSODSESCAPECHAR=23
9
10         ods escapechar="03"x;
11         %put &=SYSODSescapeChar;
SYSODSESCAPECHAR=03
12
13         ods escapechar='%';
14         %put &=SYSODSescapeChar;
SYSODSESCAPECHAR=25
15
16         ods escapechar='abc';
17         %put &=SYSODSescapeChar;
SYSODSESCAPECHAR=61
```

SYSODSGRAPHICS Automatic Macro Variable

Determines whether SAS ODS Graphics is enabled or disabled.

Type:	Automatic macro variable (read-only)

Details

The SYSODSGRAPHICS automatic macro variable determines whether SAS ODS Graphics is enabled or disabled. SYSODSGRAPHICS writes the following values to the SAS log:

- `1` if ODS Graphics is enabled
- `0` if ODS Graphics is disabled
- `-1` if an internal error occurred

SYSODSPATH Automatic Macro Variable

Contains the current Output Delivery System (ODS) pathname.

Type:	Automatic macro variable (read only)
Restriction:	The SYSODSPATH automatic macro variable exists only after an ODS or PROC TEMPLATE statement is invoked.

Details

The SYSODSPATH automatic macro variable contains the current ODS template search path.

```
ODS;
%put &sysodspath;
```

```
SASUSER.TEMPLAT(UPDATE) SASHELP.TMPLMST(READ)
```

SYSPARM Automatic Macro Variable

Contains a character string that can be passed from the operating environment to SAS program steps.

Type:	Automatic macro variable (read and write)

Details

SYSPARM enables you to pass a character string from the operating environment to SAS program steps and provides a means of accessing or using the string while a program is executing. For example, you can use SYSPARM from the operating environment to pass a title statement or a value for a program to process. You can also set the value of SYSPARM within a SAS program. SYSPARM can be used anywhere in a SAS program. The default value of SYSPARM is null (zero characters).

SYSPARM is most useful when specified at invocation of SAS. For more information, see the SAS documentation for your operating environment.

Note: The macro processor always stores the value of SYSPARM in unquoted form. To quote the resolved value of SYSPARM, use the %SUPERQ macro quoting function.

Comparisons

- Assigning a value to SYSPARM is the same as specifying a value for the SYSPARM= system option.
- Retrieving the value of SYSPARM is the same as using the SYSPARM() SAS function.

Example: Passing a Value to a Procedure

In this example, you invoke SAS on a UNIX operating environment on September 20, 2011 (the librefs Dept and Test are defined in the config.sas file) with a command like the following:

```
sas program-name -sysparm dept.projects -config /myid/config.sas
```

Macro variable SYSPARM supplies the name of the data set for PROC REPORT:

```
proc report data=&sysparm
     report=test.resorces.priority.rept;
title "%sysfunc(date(),worddate.)";
title2;
title3 'Active Projects By Priority';
run;
```

SAS sees the following:

```
proc report data=dept.projects
     report=test.resorces.priority.rept;
title "September 20, 2011";
title2;
title3 'Active Projects By Priority';
run;
```

SYSPBUFF Automatic Macro Variable

Contains text supplied as macro parameter values.

Type:	Automatic macro variable (read and write, local scope)

Details

SYSPBUFF resolves to the text supplied as parameter values in the invocation of a macro that is defined with the PARMBUFF option. For name-style invocations, this text includes the parentheses and commas. Using the PARMBUFF option and SYSPBUFF, you can define a macro that accepts a varying number of parameters at each invocation.

If the macro definition includes both a set of parameters and the PARMBUFF option, the macro invocation causes the parameters to receive values and the entire invocation list of values to be assigned to SYSPBUFF.

Note: The SYSPBUFF automatic macro variable can be modified only within the scope that it resides. Any attempt to assign a value to SYSPBUFF within an inner scope not already containing an instance of SYSPBUFF causes a new instance of SYSPBUFF to be created within that inner scope.

Example: Using SYSPBUFF to Display Macro Parameter Values

The macro PRINTZ uses the PARMBUFF option to define a varying number of parameters and SYSPBUFF to display the parameters specified at invocation.

```
%macro printz/parmbuff;
   %put Syspbuff contains: &syspbuff;
   %let num=1;
   %let color=%scan(&syspbuff,&num);
   %put;
   %do %while(&color ne);
      %put Color &num is &color..;
      %let num=%eval(&num+1);
      %let color=%scan(&syspbuff,&num);
   %end;
%mend printz;

%printz(purple,red,blue,teal)
```

When this program executes, this line is written to the SAS log:

```
Syspbuff contains: (purple,red,blue,teal)

Color 1 is purple.
Color 2 is red.
Color 3 is blue.
Color 4 is teal.
```

SYSPRINTTOLIST Automatic Macro Variable

Contains the path of the LIST file set by the PRINTTO procedure for future redirection.

Type:	Automatic macro variable (read only)

Details

The SYSPRINTTOLIST automatic macro variable contains the destination path for the LIST file set by the PRINTTO procedure in the current execution scope.

Note: If no redirection of the LIST file has occurred, then the value of the SYSPRINTTOLIST automatic macro variable is null.

SYSPRINTTOLOG Automatic Macro Variable

Contains the path of the LOG file set by the PRINTTO procedure for future redirection.

Type:	Automatic macro variable (read only)

Details

The SYSPRINTTOLOG automatic macro variable contains the destination path for the LOG file set by the PRINTTO procedure in the current execution scope.

Note: If no redirection of the LOG file has occurred, then the value of the SYSPRINTTOLOG automatic macro variable is null.

SYSPROCESSID Automatic Macro Variable

Contains the process ID of the current SAS process.

Type:	Automatic macro variable (read only)
Default:	null

Details

The value of SYSPROCESSID is a SAS internally generated 32–character hexadecimal string. The default value is null.

Example: Using SYSPROCESSID to Display the Current SAS Process ID

The following code writes the current SAS process ID to the SAS log:

```
%put &sysprocessid;
```

A process ID, such as the following, is written to the SAS log:

```
41D1B269F86C7C5F4010000000000000
```

SYSPROCESSMODE Automatic Macro Variable

Contains the name of the Compute Server run mode or server type.

Type:	Automatic macro variable (read-only)

Details

SYSPROCESSMODE is a read-only automatic macro variable, which contains the name of the current Compute Server run mode or server type:

- SAS Compute Server
- SAS Batch Mode
- SAS Line Mode
- SAS/CONNECT Session
- SAS Share Server
- SAS IntrNet Server
- SAS Pooled Workspace Server
- SAS Stored Process Server
- SAS OLAP Server
- SAS Table Server
- SAS Metadata Server

Example: Using SYSPROCESSMODE to Display the Current SAS Process Run Mode or Server Type

The following statements writes the current Compute Server run mode or server type to the SAS log:

```
%put &sysprocessmode;
```

Here is an example of what is written to the SAS log:

```
SAS Compute Server
```

SYSPROCESSNAME Automatic Macro Variable

Contains the process name of the current SAS process.

Type:	Automatic macro variable (read only)

Example: Using SYSPROCESSNAME to Display the Current SAS Process Name

The following statement writes the name of the current Compute Server process to the log:

```
%put &sysprocessname;
```

Here is an example of what is written to the SAS log:

```
Compute Server
```

SYSPROCNAME Automatic Macro Variable

Contains the name of the procedure (or DATASTEP for DATA steps) currently being processed by the SAS Language Processor.

Type:	Automatic macro variable (read only)

Details

The value of SYSPROCNAME contains the name of the procedure specified by the user in the PROC statement until a step boundary is reached.

SYSRC Automatic Macro Variable

Contains the decimal value of the exit status code that is returned by the last Linux command executed from your SAS session.

Type:	Automatic macro variable (read and write)
Default:	0

Details

The code returned by SYSRC is based on commands that you execute using the %SYSEXEC macro statement. Return codes are integers.

Note: XCMD must be enabled on the compute server to run the %SYSEXEC statement.

You can use SYSRC to check the return code of a system command before you continue with a job.

SYSSCP and SYSSCPL Automatic Macro Variable

Contains the abbreviation for your processor architecture.

Type: Automatic macro variable (read only)

Details

SYSSCP and SYSSCPL resolve to an abbreviation of the name of your operating environment. In some cases, SYSSCPL provides a more specific value than SYSSCP. You could use SYSSCP and SYSSCPL to check the operating environment to execute appropriate system commands.

The following table listsf the values for SYSSCP and SYSSCPL.

Table 14.9 *SYSSCP and SYSSCPL Values for Linux*

Platform	SYSSCP Value	SYSSCPL Value
LAX or LINUX on X64 (x86-64)	LIN X64	LINUX or Linux

SYSSCPL Automatic Macro Variable

Contains the name of the specific Linux environment that you are using.

Type: Automatic macro variable (read only)

Note: This variable contains the same value that is returned by the Linux command uname.

Details

See “SYSSCP and SYSSCPL Automatic Macro Variable” on page 270

SYSSITE Automatic Macro Variable

Contains the number assigned to your site.

Type:	Automatic macro variable (read only)

Details

SAS assigns a site number to each site that licenses SAS software. The number is displayed in the SAS log.

SYSSIZEOFLONG Automatic Macro Variable

Contains the length in bytes of a long integer in the current session.

Type:	Automatic macro variable (read only)

Details

The SYSSIZEOFLONG automatic macro variable contains the length of a long integer in the current Compute Server session.

SYSSIZEOFPTR Automatic Macro Variable

Contains the size in bytes of a pointer.

Type:	Automatic macro variable (read only)

Details

The SYSSIZEOFPTR automatic macro variable contains the size in bytes of a pointer.

SYSSIZEOFUNICODE Automatic Macro Variable

Contains the length in bytes of a Unicode character in the current session.

Type:	Automatic macro variable (read only)

Details

The SYSSIZEOFUNICODE automatic macro variable contains the length of the Unicode character in the current Compute Server session.

SYSSTARTID Automatic Macro Variable

Contains the ID generated from the last STARTSAS statement (experimental).

Type:	Automatic macro variable (read only)
Default:	null
Note:	STARTSAS statement is an experimental feature of the SAS System.

Details

The ID is a 32-character hexadecimal string that can be passed to the WAITSAS statement or the ENDSAS statement. The default value is null.

Example: Using SYSSTARTID to Display the SAS Process ID from the Most Recent STARTSAS Statement (experimental)

Submit the following code from the SAS process in which you have submitted the most recent STARTSAS statement to write the value of the SYSSTARTID variable to the SAS log:

```
%put &sysstartid;
```

A process ID value, such as the following, is written to the SAS log:

```
41D20425B89FCED94036000000000000
```

SYSSTARTNAME Automatic Macro Variable

Contains the process name generated from the last STARTSAS statement (experimental).

Type:	Automatic macro variable (read only)
Default:	null
Note:	STARTSAS statement is an experimental feature of the SAS System.

Example: Using SYSSTARTNAME to Display the SAS Process Name from the Most Recent STARTSAS Statement (experimental)

Submit the following code from the SAS process in which you have submitted the most recent STARTSAS statement to write the value of the SYSSTARTNAME variable to the SAS log:

```
%put &sysstartname;
```

An example of a process name that can appear in the SAS log is as follows:

```
DMS Process (2)
```

SYSTCPIPHOSTNAME Automatic Macro Variable

Contains the host names of the local and remote computers when multiple TCP/IP stacks are supported.

Type:	Automatic macro variable (read only)

Details

SYSTCPIPHOSTNAME contains the host name of the system that is running multiple TCPIP stacks. If you are running a single TCPIP stack, use the SYSHOSTNAME automatic macro variable. For more information about TCPIP stacks, see your SAS host companion documentation.

See Also

"SYSHOSTNAME Automatic Macro Variable" on page 251

SYSTIME Automatic Macro Variable

Contains the time at which a SAS job or session began executing.

Type:	Automatic macro variable (read only)

Details

The value is displayed in TIME5. format and does not change during the individual job or session.

Example: Using SYSTIME to Display the Time That a Compute Server Session Started

The following statement displays the time at which a Compute Server session started.

```
%put This session started running at: &systime;
```

When this statement executes at 3 p.m., but your Compute Server session began executing at 9:30 a.m., the following comment is written to the SAS log:

```
This session started running at: 09:30
```

SYSTIMEZONE Automatic Macro Variable

Contains the time zone name based on TIMEZONE option.

Type:	Automatic macro variable (read only)

Details

SYSTIMEZONE contains the time zone name based on the current value of the TIMEZONE option. For more information about the TIMEZONE option, see the SAS documentation for your operating environment.

Example Code 14.1 *Using SYSTIMEZONE*

```
option timezone='america/new_york';
%put &=systimezone;

option timezone='america/chicago';
%put &=systimezone;

option timezone='america/denver';
%put &=systimezone;

option timezone='america/los_angeles';
%put &=systimezone;
```

Example Code 14.1 *Output*

```
114   option timezone='america/new_york';
115   %put &=systimezone;
SYSTIMEZONE=EDT
116   option timezone='america/chicago';
117   %put &=systimezone;
SYSTIMEZONE=CDT
118   option timezone='america/denver';
119   %put &=systimezone;
SYSTIMEZONE=MDT
120   option timezone='america/los_angeles';
121   %put &=systimezone;
SYSTIMEZONE=PDT
```

SYSTIMEZONEIDENT Automatic Macro Variable Automatic Macro Variable

Contains the time zone ID based on the TIMEZONE= system option.

Type:	Automatic macro variable (read only)

Details

SYSTIMEZONEIDENT contains the time zone ID based on the current value of the TIMEZONE= system option. For more information about the TIMEZONE= option, see the SAS documentation for your operating environment.

Example Code 14.2 *Using SYSTIMEZONEIDENT*

```
option timezone='america/new_york';
%put &=systimezoneident;

option timezone='america/chicago';
%put &=systimezoneident;
```

```
 option timezone='america/denver';
%put &=systimezoneident;

option timezone='america/los_angeles';
%put &=systimezoneident;
```

Example Code 14.2 *Output*

```
12   option timezone='america/new_york';
13   %put &=systimezoneident;
SYSTIMEZONEIDENT=AMERICA/NEW_YORK
14
15   option timezone='america/chicago';
16   %put &=systimezoneident;
SYSTIMEZONEIDENT=AMERICA/CHICAGO
17
18   option timezone='america/denver';
19   %put &=systimezoneident;
SYSTIMEZONEIDENT=AMERICA/DENVER
20
21   option timezone='america/los_angeles';
22   %put &=systimezoneident;
SYSTIMEZONEIDENT=AMERICA/LOS_ANGELES
```

SYSTIMEZONEOFFSET Automatic Macro Variable

Contains the current time zone offset based on TIMEZONE option.

Type:	Automatic Macro Variable

Details

SYSTIMEZONEOFFSET contains the time zone offset based on the current value of the TIMEZONE option. For more information about the TIMEZONE option, see the SAS documentation for your operating environment.

Example Code 14.3 *Using SYSTIMEZONEOFFSET*

```
option timezone='america/new_york';
%put &=systimezoneoffset;

option timezone='america/chicago';
%put &=systimezoneoffset;

option timezone='america/denver';
%put &=systimezoneoffset;

option timezone='america/los_angeles';
%put &=systimezoneoffset;
```

Example Code 14.3 *Output*

```
153   option timezone='america/new_york';
154   %put &=systimezoneoffset;
SYSTIMEZONEOFFSET=-14400
155   option timezone='america/chicago';
156   %put &=systimezoneoffset;
SYSTIMEZONEOFFSET=-18000
157   option timezone='america/denver';
158   %put &=systimezoneoffset;
SYSTIMEZONEOFFSET=-21600
159   option timezone='america/los_angeles';
160   %put &=systimezoneoffset;
SYSTIMEZONEOFFSET=-25200
```

SYSUSERID Automatic Macro Variable

Contains the user ID or login of the current SAS process.

Type:	Automatic macro variable (read only)

Example: Using SYSUSERID to Display the User ID for the Current SAS Process

The following code, when submitted from the current SAS process, writes the user ID or login for the current SAS process to the SAS log:

```
%put &sysuserid;
```

A user ID, such as the following, is written to the SAS log:

```
MyUserid
```

SYSVER Automatic Macro Variable

Contains the release number of SAS software that is running.

Type:	Automatic macro variable (read only)
See:	“SYSVLONG Automatic Macro Variable” on page 280 and “SYSVLONG4 Automatic Macro Variable” on page 280

Comparisons

SYSVER provides the release number of the SAS software that is running. You can use SYSVER to check for the release of SAS before running a job with newer features.

Example: Identifying SAS Software Release

The following statement displays the release number of a user's SAS software:

```
%put I am using release &sysver.;
```

Submitting this statement writes the following to the SAS log:

```
I am using release V.04.00.
```

SYSVIYARELEASE Automatic Macro Variable

Contains the SAS Viya platform cadence release number.

Type:	Automatic macro variable (read only)

Details

The SYSVIYARELEASE automatic macro variable contains the SAS Viya platform cadence release. The cadence release string format is as follows:

```
YYYYmmdd.nnnnnnnnnnnnn
```

Example

Display the SAS Viya platform release:

```
%put SAS Viya platform release: &sysviyarelease;
```

Here is an example of the output.

```
SAS Viya platform release: 20230901.1693566968705
```

SYSVIYAVERSION Automatic Macro Variable

Contains the SAS Viya platform cadence version.

Type:	Automatic macro variable (read only)

Details

About SAS Viya Platform Cadence Versions

The SAS Viya platform release cadence consists of Stable and Long-Term Support (LTS) releases. Stable releases typically have a cadence of one month, and LTS releases have a cadence of approximately six months. The cadence version is stored in automatic macro variable SYSVIYAVERSION.

The SAS Viya platform version strings have a specific format. The format for Stable releases changed starting with 2022.09. The following sections provide details.

Cadence Version Format for 2022.1.4 and Prior Releases

For a Stable release, the cadence version string format for 2022.1.4 and prior releases is:

```
Stable YYYY.n.n
```

where *YYYY* is the release year and *n.n* is the major and minor release numbers, respectively. Here is an example:

```
Stable 2020.0.6
```

For an LTS release, the cadence version string format is:

```
LTS YYYY.n
```

where *YYYY* is the release year and *n* is the LTS release number for that year. Here is the cadence version for the first LTS release in the year 2020:

```
LTS 2020.1
```

Cadence Version Format for 2022.09 and Later Releases

Starting with 2022.09, the Stable release version string format is:

```
Stable YYYY.nn
```

where *YYYY* is the release year and *nn* is the release month in the release year. Here is the cadence version for the 2023 May Stable release:

```
Stable 2023.05
```

The LTS release cadence version string format is:

```
LTS YYYY.nn
```

where *YYYY* is the release year and *nn* is the release month in the release year. Here is cadence version for the 2023 March LTS release:

```
LTS 2023.03
```

Example

Display the SAS Viya platform version:

```
%put SAS Viya platform version: &sysviyaversion;
```

Here is an example of the output.

```
SAS Viya platform version: LTS 2023.03
```

SYSVLONG Automatic Macro Variable

Contains the release number and maintenance level of SAS software that is running.

Type:	Automatic macro variable (read only)
See:	"SYSVER Automatic Macro Variable" on page 277 and "SYSVLONG4 Automatic Macro Variable" on page 280

Comparisons

SYSVLONG provides the release number and maintenance level of SAS software, in addition to the release number.

Example: Identifying a Maintenance Release

The following statement writes the release number and maintenance level of the SAS software to the SAS log:

```
%put I am using release: &sysvlong..;
```

Here is an example of the output:

```
I am using release: V.04.00M0P083123.
```

SYSVLONG4 Automatic Macro Variable

Contains the release number and maintenance level of SAS software that is running and has a four-digit year.

Type:	Automatic macro variable (read only)
See:	"SYSVER Automatic Macro Variable" on page 277 and "SYSVLONG Automatic Macro Variable" on page 280

Comparisons

SYSVLONG4 provides a four-digit year and the release number and maintenance level of SAS software. SYSVLONG does not contain the four-digit year but everything else is the same.

Example: Using SYSVLONG4 Automatic Macro Variable

The following statement displays information that identifies the release being used.

```
%put I am using maintenance release: &sysvlong4;
```

Here is an example of the output:

```
I am using maintenance release: V.04.00M0P12222022
```

Here is an explanation for V.04.00M0P12222022:

V.04.00
: SAS Viya platform version 4.00

M0
: Maintenance level 0

P
: Performance.

12222022
: The software was built on December 22nd, 2022.

SYSWARNINGTEXT Automatic Macro Variable

Contains the text of the last warning message formatted for display in the SAS log.

Type:	Automatic macro variable (read and write starting with 2024.05, read only in 2024.04 and prior releases)

Details

The value of SYSWARNINGTEXT is the text of the last warning message generated in the SAS log. For a list of SYSERR warnings and errors, see "SYSERR Automatic Macro Variable" on page 247.

Note: If the last warning message text that was generated contains an `&` or `%` and you are using the %PUT statement, you must use the %SUPERQ macro quoting function to mask the special characters to prevent further resolution of the value. The following example uses the %PUT statement and the %SUPERQ macro quoting function:

```
%put %superq(syswarningtext);
```

For more information, see "%SUPERQ Macro Function" on page 347.

Example: Using SYSWARNINGTEXT

This example creates a warning message:

```
data NULL;
    set doesnotexist;
run;
%put &syswarningtext;
```

When these statements execute, the following comments are written to the SAS log:

```
 1  data NULL;
 2  set doesnotexist;
 ERROR: File WORK.DOESNOTEXIST.DATA does not exist.
 3  run;
 NOTE: The SAS System stopped processing this step because of errors.
 WARNING: The data set WORK.NULL might be incomplete.  When this step
          was stopped there were 0 observations and 0 variables.
 NOTE: DATA statement used (Total process time):
     real time           11.16 seconds
     cpu time            0.07 seconds
 4  %put &syswarningtext;
The data set WORK.NULL might be incomplete.  When this step was
stopped there were 0 observations and 0 variables.
```

15

DATA Step Call Routines for Macros

DATA Step Call Routines for Macros

You can interact with the macro facility using DATA step call routines.

Dictionary

CALL EXECUTE Routine

Resolves the argument, and issues the resolved value for execution at the next step boundary.

Type:	DATA step call routine

Syntax

CALL EXECUTE(*argument*);

Required Argument

argument

can be one of the following:

- a character string, enclosed in quotation marks. *Argument* within single quotation marks resolves during program execution. *Argument* within double quotation marks resolves while the DATA step is being constructed. For example, to invoke the macro SALES, you can use the following code:

```
call execute('%sales');
```

- the name of a DATA step character variable whose value is a text expression or a SAS statement to be generated. Do not enclose the name of the DATA step variable in quotation marks. For example, to use the value of the DATA step variable FINDOBS, which contains a SAS statement or text expression, you can use the following code:

```
call execute(findobs);
```

- a character expression that is resolved by the DATA step to a macro text expression or a SAS statement. For example, to generate a macro invocation whose parameter is the value of the variable MONTH, you use the following code:

```
call execute('%sales('||month||')');
```

Note System option SOURCE controls whether the code that is passed to the CALL EXECUTE routine is written to the SAS log. When SOURCE is in effect, the code is written to the log. Otherwise, it is suppressed. For batch jobs, the default typically is SOURCE. Use the OPTIONS procedure to see the current setting and an OPTIONS statement to change it:

```
proc options option=source; run; /* Print current setting */
options nosource;                /* Do not write source to the SAS log */
```

See "Example 3: How to Create a Macro Using CALL EXECUTE" on page 287.

See "OPTIONS Procedure" in *SAS System Options: Reference*

"SOURCE System Option" in *SAS System Options: Reference*

Details

If an EXECUTE routine argument is a macro invocation or resolves to one, the macro executes immediately. Execution of SAS statements generated by the execution of the macro will be delayed until after a step boundary. SAS macro statements, including macro variable references, execute immediately.

Note: Because of the delay of the execution of the SAS statements until after a step boundary, references in SAS macro statements to macro variables created or updated by the SAS statements will not resolve properly.

Note: 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. For a workaround, see the following TIP.

TIP The following example uses the %NRSTR macro quoting function to mask the macro statement. This function will delay the execution of macro statements until after a step boundary.

```
call execute('%nrstr(%sales('||month||'))');
```

Comparisons

Unlike other elements of the macro facility, a CALL EXECUTE statement is available regardless of the setting of the SAS system option MACRO | NOMACRO. In both cases, EXECUTE places the value of its argument in the program stack. However, when NOMACRO is set, any macro calls or macro functions in the argument are not resolved.

Examples

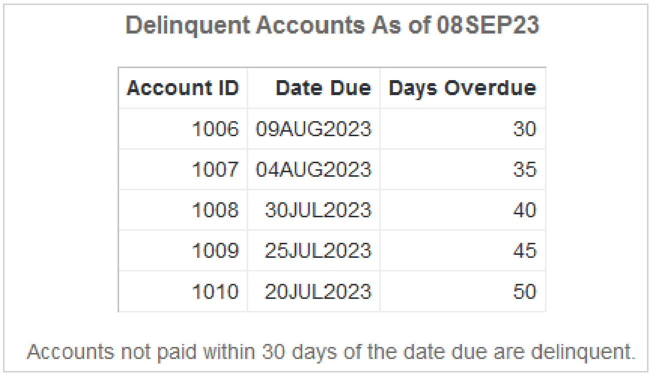
Example 1: Executing a Macro Conditionally

In the following code, the first DATA step creates test data to use in this example. Macro %OVERDUE is defined, which outputs data set Late. The second DATA step uses CALL EXECUTE to execute macro %OVERDUE only if the DATA step writes at least one observation to data set Late.

```
data Work.Billed;
   retain dateDue;
   dateDue = today();
   do accountID = 1000 to 1010;
      output;
      dateDue = dateDue - 5;
   end;
   format dateDue Date9.;
   label dateDue="Date Due" accountID = "Account ID";
run;

%macro overdue;
   proc print data=late(drop=overdueAccounts) label noobs;
```

```
      title "Delinquent Accounts As of &sysdate";
      footnote "Accounts not paid within 30 days of the date due are 
delinquent.";
      var accountID dateDue daysOverdue;
   run;
   title;
   footnote;
%mend overdue;

data late;
   retain overdueAccounts 0;
   set Work.Billed end=final;
   if dateDue<=today()-30 then
      do;
         daysOverdue = today() - dateDue;
         overdueAccounts=1;
         output;
      end;
   label daysOverdue="Days Overdue";
   if final and overdueAccounts then call execute('%overdue');
run;
```

Here is an example of the output from macro %OVERDUE.



Delinquent Accounts As of 08SEP23

Account ID	Date Due	Days Overdue
1006	09AUG2023	30
1007	04AUG2023	35
1008	30JUL2023	40
1009	25JUL2023	45
1010	20JUL2023	50

Accounts not paid within 30 days of the date due are delinquent.

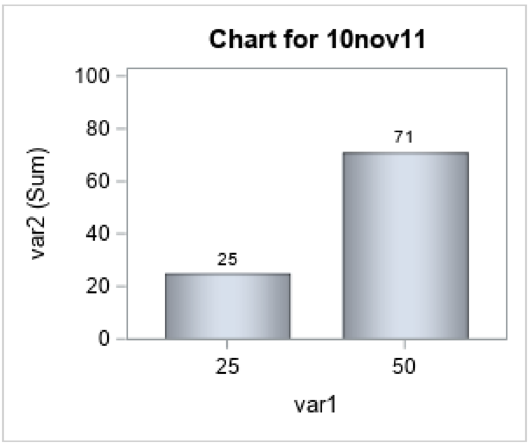
Example 2: Passing DATA Step Values into a Parameter List

In the following code, CALL EXECUTE passes the value of the DATE variable in the Dates data set to macro REPT for its DAT parameter, variable VAR1 in the REPTDATA data set for its CAT parameter, variable VAR2 in the REPTDATA data set for its RSP parameter, and REPTDATA as the value for its DSN parameter. After the DATA _NULL_ step finishes, three PROC SGPLOT statements are submitted, one for each of the three dates in the Dates data set.

```
data dates;
   input date $;
datalines;
10nov11
11nov11
12nov11
;
data reptdata;
   input date $ var1 var2;
datalines;
10nov11 25 25
```

```
10nov11 50 71
11nov11 23 90
11nov11 30 29
12nov11 33 44
12nov11 75 86
;
options source;
%macro rept(dat,cat,rsp,dsn);
   ods graphics / width=300px height=250px scale=off;
   proc sgplot data=&dsn;
       title "Chart for &dat";
       where date eq "&dat";
       vbar &cat / response=&rsp datalabel dataskin=pressed
barwidth=0.7;
       yaxis min=0 max=100;
   run;
%mend rept;
data _null_;
   set dates;
   call execute('%rept('||date||',var1,var2,reptdata)');
run;
```

Here is the output for the first PROC SGPLOT statement, which is for date 10nov11.



For information about the SGPLOT procedure, see "SGPLOT Procedure" in *SAS ODS Graphics: Procedures Guide*.

Example 3: How to Create a Macro Using CALL EXECUTE

The following code creates the macro P using the CALL EXECUTE routine. In this example, system option SOURCE is specified so that the code generated by the CALL EXECUTE routine is written to the SAS log.

```
options source;
data _null_;
   call execute('%nrstr(%%)macro p;

proc print data=sashelp.class;
run;

%nrstr(%%)mend;');
run;
```

```
%p
```

When this program is executed, the following note is written to the SAS log showing the code that was generated by the CALL EXECUTE routine.

```
NOTE: CALL EXECUTE generated line.
1   + %macro p;proc print data=sashelp.class;run;%mend;
```

Note: To suppress the generated code in the SAS log, specify system option NOSOURCE.

CALL SYMDEL Routine

Deletes the specified variable from the macro global symbol table.

Type: DATA step call routine

Syntax

CALL SYMDEL(*macro-variable*<, *option*>);

Required Arguments

macro-variable
can be any of the following:

- the name of a macro variable within quotation marks but without an ampersand. When a macro variable value contains another macro variable reference, SYMDEL does not attempt to resolve the reference.
- the name of a DATA step character variable, specified with no quotation marks, which contains the name of a macro variable. If the value is not a valid SAS name, or if the macro processor cannot find a macro variable of that name, SAS writes a warning to the log.
- a character expression that constructs a macro variable name.

option

NOWARN
suppresses the warning message when an attempt is made to delete a non-existent macro variable. NOWARN must be within quotation marks.

Details

CALL SYMDEL issues a warning when an attempt is made to delete a non-existent macro variable. To suppress this message, use the NOWARN option.

CALL SYMPUT Routine

Assigns a value produced in a DATA step to a macro variable.

Category:	Macro
Type:	DATA step call routine
Restriction:	The SYMPUT CALL routine is not supported by the CAS engine.
See:	"SYMGET Function" on page 303 and "CALL SYMPUTX Routine" on page 295

Syntax

CALL SYMPUT(*macro-variable*, *value*);

Required Arguments

macro-variable

can be one of the following items:

- a character string that is a SAS name, enclosed in quotation marks. For example, to assign the character string `testing` to macro variable NEW, submit the following statement:

```
call symput('new','testing');
```

- the name of a character variable whose values are SAS names. For example, this DATA step creates the three macro variables SHORTSTP, PITCHER, and FRSTBASE and respectively assigns them the values ANN, TOM, and BILL.

```
data team1;
   input position : $8. player : $12.;
   call symput(position,player);
datalines;
shortstp Ann
pitcher Tom
frstbase Bill
;

%put Short stop: &shortstp;
%put Pitcher:    &pitcher;
%put First Base: &frstbase;
```

- a character expression that produces a macro variable name. This form is useful for creating a series of macro variables. For example, the CALL SYMPUT statement builds a series of macro variable names by combining the character string POS and the left-aligned value of _N_ . Values are assigned to the macro variables POS1, POS2, and POS3.

```
data team2;
   input position : $12. player $12.;
   call symput('POS'||left(_n_), position);
   datalines;
shortstp Ann
pitcher Tom
frstbase Bill
;

%put Position 1: &pos1;
%put Position 2: &pos2;
%put Position 3: &pos3;
```

value

is the value to be assigned, which can be

- a string enclosed in quotation marks. For example, this statement assigns the string `testing` to the macro variable NEW:

```
call symput('new','testing');
```

- the name of a numeric or character variable. The current value of the variable is assigned as the value of the macro variable. If the variable is numeric, SAS performs an automatic numeric-to-character conversion and writes a message in the log. Later sections on formatting rules describe the rules that SYMPUT follows in assigning character and numeric values of DATA step variables to macro variables.

  Note: This form is most useful when *macro-variable* is also the name of a SAS variable or a character expression that contains a SAS variable. A unique macro variable name and value can be created from each observation, as shown in the previous example for creating the data set Team1.

  If *macro-variable* is a character string, SYMPUT creates only one macro variable, and its value changes in each iteration of the program. Only the value assigned in the last iteration remains after program execution is finished.

- a DATA step expression. The value returned by the expression in the current observation is assigned as the value of *macro-variable*. In this example, the macro variable named HOLDATE receives the value `July 4,1997`:

```
data c;
   input holiday mmddyy.;
   call symput('holdate',trim(left(put(holiday,worddate.))));
datalines;
070497
;
run;
```

  If the expression is numeric, SAS performs an automatic numeric-to-character conversion and writes a message in the log. Later sections on formatting rules describe the rules that SYMPUT follows in assigning character and numeric values of expressions to macro variables.

Details

If *macro-variable* exists in any enclosing scope, *macro-variable* is updated. If *macro-variable* does not exist, SYMPUT creates it. (See below to determine in which scope SYMPUT creates *macro-variable*.) SYMPUT makes a macro variable assignment when the program executes.

SYMPUT can be used in all SAS language programs, including SCL programs. Because it resolves variables at program execution instead of macro execution, SYMPUT should be used to assign macro values from DATA step views, SQL views, and SCL programs.

Scope of Variables Created with SYMPUT

SYMPUT puts the macro variable in the most local symbol table that it finds. If there are no local tables, then the macro variable is put in the global symbol table.

For more information about creating a variable with SYMPUT, see "Scopes of Macro Variables" on page 57.

Note: 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. For a workaround, see the following TIP:

TIP If CALL SYMPUT is contained within a macro and that macro is invoked within a CALL EXECUTE routine, the macro variable will not resolve inside the macro. To get around this issue, invoke the macro in one of these two ways:

```
call execute('%nrstr(%test('||temp||'))');
```

Use the %NRSTR macro quoting function to mask the macro statement.

```
rc=dosubl('%test('||temp||')');
```

Problem Trying to Reference a SYMPUT-Assigned Value Before It Is Available

One of the most common problems in using SYMPUT is trying to reference a macro variable value assigned by SYMPUT before that variable is created. The failure generally occurs because the statement referencing the macro variable compiles before execution of the CALL SYMPUT statement that assigns the variable's value. The most important fact to remember in using SYMPUT is that it assigns the value of the macro variable during program execution. Macro variable references resolve during the compilation of a step, a global statement used outside a step, or an SCL program. As a result:

- You cannot use a macro variable reference to retrieve the value of a macro variable in the same program (or step) in which SYMPUT creates that macro variable and assigns it a value.
- You must specify a step boundary statement to force the DATA step to execute before referencing a value in a global statement following the program (for example, a TITLE statement). The boundary could be a RUN statement or another DATA or PROC statement. For example:

```
data x;
   x='December';
   call symput('var',x);
proc print;
title "Report for &var";
run;
```

Processing on page 45 provides details about compilation and execution.

Formatting Rules For Assigning Character Values

If *value* is a character variable, SYMPUT writes it using the $*w*. format, where *w* is the length of the variable. Therefore, a value shorter than the length of the program variable is written with trailing blanks. For example, in the following DATA step the length of variable C is 8 by default. Therefore, SYMPUT uses the $8. format and assigns the letter `x` followed by seven trailing blanks as the value of CHAR1. To eliminate the blanks, use the TRIM function as shown in the second SYMPUT statement.

```
data char1;
   input c $;
   call symput('char1',c);
   call symput('char2',trim(c));
   datalines;
x
;
run;
%put char1 = ***&char1***;
%put char2 = ***&char2***;
```

When this program executes, these lines are written to the SAS log:

```
char1 = ***x       ***
char2 = ***x***
```

Formatting Rules For Assigning Numeric Values

If *value* is a numeric variable, SYMPUT writes it using the BEST12. format. The resulting value is a 12-byte string with the value right-aligned within it. For example, this DATA step assigns the value of numeric variable X to the macro variables NUM1 and NUM2. The last CALL SYMPUT statement deletes undesired leading blanks by using the LEFT function to left-align the value before the SYMPUT routine assigns the value to NUM2.

```
data _null_;
   x=1;
   call symput('num1',x);
   call symput('num2',left(x));
   call symput('num3',trim(left(put(x,8.)))); /*preferred technique*/
```

```
run;
%put num1 = ***&num1***;
%put num2 = ***&num2***;
%put num3 = ***&num3***;
```

When this program executes, these lines are written to the SAS log:

```
num1 = ***           1***
num2 = ***1           ***
num3 = ***1***
```

Comparisons

- SYMPUT assigns values produced in a DATA step to macro variables during program execution, but the SYMGET function returns values of macro variables to the program during program execution.
- SYMPUT is available in DATA step and SCL programs, but SYMPUTN is available only in SCL programs.
- SYMPUT assigns character values, but SYMPUTN assigns numeric values.

Example: Creating Macro Variables and Assigning Them Values from a Data Set

```
data dusty;
   input dept $ name $ salary @@;
   datalines;
bedding Watlee 18000    bedding Ives 16000
bedding Parker 9000     bedding George 8000
bedding Joiner 8000     carpet Keller 20000
carpet Ray 12000        carpet Jones 9000
gifts Johnston 8000     gifts Matthew 19000
kitchen White 8000      kitchen Banks 14000
kitchen Marks 9000      kitchen Cannon 15000
tv Jones 9000           tv Smith 8000
tv Rogers 15000         tv Morse 16000
;
proc means noprint;
   class dept;
   var salary;
   output out=stats sum=s_sal;
run;
data _null_;
   set stats;
   if _n_=1 then call symput('s_tot',trim(left(s_sal)));
   else call symput('s'||dept,trim(left(s_sal)));
run;
%put _user_;
```

When this program executes, this list of variables is written to the SAS log:

```
GLOBAL SCARPET 41000
GLOBAL SKITCHEN 46000
GLOBAL STV 48000
GLOBAL SGIFTS 27000
GLOBAL SBEDDING 59000
GLOBAL S_TOT 221000
```

CALL SYMPUTN Routine

In SCL programs, assigns a numeric value to a global macro variable.

Category:	Macro
Type:	SCL call routine
Restriction:	The SYMPUT CALL routine is not supported by the CAS engine.
See:	"SYMGET Function" on page 303, "SYMGETN Function" on page 307, and "CALL SYMPUT Routine" on page 289

Syntax

CALL SYMPUTN('*macro-variable*', *value*);

Required Arguments

macro-variable
is the name of a global macro variable with no ampersand – note the single quotation marks. Or, it is the name of an SCL variable that contains the name of a global macro variable.

value
is the numeric value to assign, which can be a number or the name of a numeric SCL variable.

Details

The SYMPUTN routine assigns a numeric value to a global SAS macro variable. SYMPUTN assigns the value when the SCL program executes. You can also use SYMPUTN to assign the value of a macro variable whose name is stored in an SCL variable. For example, to assign the value of SCL variable UNITNUM to SCL variable UNITVAR, which contains 'UNIT', submit the following:

```
call symputn(unitvar,unitnum)
```

You must use SYMPUTN with a CALL statement.

Note: It is inefficient to use an ampersand (&) to reference a macro variable that was created with CALL SYMPUTN. Instead, use SYMGETN. It is also inefficient to use CALL SYMPUTN to store a variable that does not contain a numeric value.

Comparisons

- SYMPUTN assigns numeric values, but SYMPUT assigns character values.
- SYMPUTN is available only in SCL programs, but SYMPUT is available in DATA step programs and SCL programs.
- SYMPUTN assigns numeric values, but SYMGETN retrieves numeric values.

Example: Storing the Value 1000 in the Macro Variable UNIT When the SCL Program Executes

This statement stores the value 1000 in the macro variable UNIT when the SCL program executes:

```
call symputn('unit',1000);
```

CALL SYMPUTX Routine

Assigns a value to a macro variable, and removes both leading and trailing blanks.

Category:	Macro
Restriction:	The SYMPUT CALL routine is not supported by the CAS engine.
See:	"CALL SYMPUTX Routine" in *SAS Functions and CALL Routines: Reference*

Syntax

CALL SYMPUTX(*macro-variable*, *value* <, *symbol-table*>);

Required Arguments

macro-variable
can be one of the following items:

- a character string that is a SAS name, enclosed in quotation marks.
- the name of a character variable whose values are SAS names.

- a character expression that produces a macro variable name. This form is useful for creating a series of macro variables.
- a character constant, variable, or expression. Leading and trailing blanks are removed from the value of name, and the result is then used as the name of the macro variable.

value
is the value to be assigned, which can be

- a string enclosed in quotation marks.
- the name of a numeric or character variable. The current value of the variable is assigned as the value of the macro variable. If the variable is numeric, SAS performs an automatic numeric-to-character conversion and writes a message in the log.

  Note: This form is most useful when *macro-variable* is also the name of a SAS variable or a character expression that contains a SAS variable. A unique macro variable name and value can be created from each observation.

- a DATA step expression. The value returned by the expression in the current observation is assigned as the value of *macro-variable*.

  If the expression is numeric, SAS performs an automatic numeric-to-character conversion and writes a message in the log.

Optional Argument

symbol-table
specifies a character constant, variable, or expression. The value of *symbol-table* is not case sensitive. The first non-blank character in *symbol-table* specifies the symbol table in which to store the macro variable. The following values are valid as the first non-blank character in *symbol-table*:

G
specifies that the macro variable is stored in the global symbol table, even if the local symbol table exists.

L
specifies that the macro variable is stored in the most local symbol table that exists, which will be the global symbol table, if used outside a macro.

F
specifies that if the macro variable exists in any symbol table, CALL SYMPUTX uses the version in the most local symbol table in which it exists. If the macro variable does not exist, CALL SYMPUTX stores the variable in the most local symbol table.

Note: If you omit *symbol-table*, leave it blank or use any other symbol other than G, L, or F. CALL SYMPUTX stores the macro variable in the same symbol table as does the CALL SYMPUT routine.

Comparisons

CALL SYMPUTX is similar to CALL SYMPUT. Here are the differences.

- CALL SYMPUTX does not write a note to the SAS log when the second argument is numeric. CALL SYMPUT, however, writes a note to the log stating that numeric values were converted to character values.
- CALL SYMPUTX uses a field width of up to 32 characters when it converts a numeric second argument to a character value. CALL SYMPUT uses a field width of up to 12 characters.
- CALL SYMPUTX left-justifies both arguments and trims trailing blanks. CALL SYMPUT does not left-justify the arguments, and trims trailing blanks from the first argument only. Leading blanks in the value of name cause an error.
- CALL SYMPUTX enables you to specify the symbol table in which to store the macro variable, whereas CALL SYMPUT does not.

Example

The following example shows the results of using CALL SYMPUTX.

```
%let x=1;
%let items=2;
%macro test(val);

data _null_;
 call symputx('items', ' leading and trailing blanks removed ', 'L');
 call symputx('  x   ', 123.456);
run;

%put local items &items;
%mend test;

%test(100)

%put items=!&items!;
%put x=!&x!;
```

The following lines are written to the SAS log:

```
82   %let x=1;
83   %let items=2;
84   %macro test(val);
85
86   data _null_;
87    call symputx('items', ' leading and trailing blanks removed ', 'L');
88    call symputx('  x   ', 123.456);
89   run;
90
91   %put local items &items
92   %mend test;
93
94   %test(100)
NOTE: DATA statement used (Total process time):
      real time           0.00 seconds
      cpu time            0.00 seconds

local items leading and trailing blanks removed
95
96   %put items=!&items!;
items=!2!
97   %put x=!&x!;
x=!123.456!
```

16

DATA Step Functions for Macros

DATA Step Functions for Macros

You can interact with the macro facility using DATA step functions.

Dictionary

RESOLVE Function

Resolves the value of a text expression during DATA step execution.

Type:	DATA step function

Syntax

RESOLVE(*argument*)

Required Argument

argument

specifies a text expression to be resolved. *Argument* can be one of the following:

'text-expression'

specifies a text expression within quotation marks.

A text expression prevents the macro processor from resolving the argument while the DATA step is being constructed. When a macro variable value contains a macro variable reference, RESOLVE attempts to resolve the reference. If *text-expression* references a nonexistent macro variable, RESOLVE returns the unresolved reference.

Requirement *Text-expression* must be enclosed in quotation marks.

Note *Text-expression* should not reference a secure macro.

Examples This example uses a text expression to assign the text generated by macro %LOCATE to variable X.

```
x=resolve('%locate');
```

This example uses a text expression to assign the value of macro variable Name to variable X.

```
x=resolve('&name');
```

character-variable-name

specifies the name of a DATA step variable that contains a text expression.

Note *Character-variable-name* should not reference a secure macro.

Example This example assigns the value of the text expression stored in DATA step variable Addr1 to X:

```
addr1='&locate';
x=resolve(addr1);
```

character-expression

specifies a character expression that produces a text expression for resolution by the macro facility.

Note *Character-expression* should not reference a secure macro.

Example This example uses the current value of the DATA step variable StNum in building the name of a macro:

```
x=resolve('%state'||left(stnum));
```

Details

The RESOLVE function returns a character value that is the maximum length of a DATA step character variable unless you specifically assign the target variable a shorter length. A returned value that is longer is truncated.

If RESOLVE cannot locate the macro variable or macro identified by the argument, it returns the argument without resolution and the macro processor issues a warning message.

You can create a macro variable with the SYMPUT routine and use RESOLVE to resolve it in the same DATA step.

Comparisons

- RESOLVE resolves the value of a text expression during execution of a DATA step or SCL program. Whereas a macro variable reference resolves when a DATA step is being constructed or an SCL program is being compiled. For this reason, the resolved value of a macro variable reference is constant during execution of a DATA step or SCL program. However, RESOLVE can return a different value for a text expression in each iteration of the program.
- RESOLVE accepts a wider variety of arguments than the SYMGET function accepts. SYMGET resolves only a single macro variable but RESOLVE resolves any macro expression. Using RESOLVE might result in the execution of macros and resolution of more than one macro variable.
- When a macro variable value contains an additional macro variable reference, RESOLVE attempts to resolve the reference, but SYMGET does not.
- If *argument* references a nonexistent macro variable, RESOLVE returns the unresolved reference, whereas SYMGET returns a missing value.
- Because of its greater flexibility, RESOLVE requires slightly more computer resources than SYMGET.

Example: Resolving Sample References

This example shows RESOLVE used with a macro variable reference, a macro invocation, and a DATA step variable whose value is a macro invocation.

```
%let event=Holiday;
%macro date;
   New Year
%mend date;
data test;
   length var1-var3 $ 15;
   when='%date';
   var1=resolve('&event'); /* macro variable reference */
   var2=resolve('%date');  /* macro invocation */
   var3=resolve(when);     /* DATA step variable with macro invocation
*/
```

```
   put var1=  var2=  var3=;
run;
```

When this program executes, these lines are written to the SAS log:

```
VAR1=Holiday  VAR2=New Year  VAR3=New Year
NOTE: The data set WORK.TEST has 1 observations and 4 variables.
```

SYMEXIST Function

Returns an indication of the existence of a macro variable.

Type: DATA step function

Syntax

SYMEXIST(*argument*)

Required Argument

argument
specifies the name of a macro variable. *Argument* can be one of the following:

'macro-variable-name'
specifies within quotation marks the name of a macro variable without the leading ampersand.

Example This example checks for the existence of macro variable State and assigns the result to variable X:

```
x=symexist('state');
```

character-variable-name
specifies the name of a DATA step character variable that contains the name of a macro variable without the leading ampersand.

Example This example assigns the name of macro variable Locate to character variable Addr1, checks for the existence of the macro variable contained in variable Addr1, and assigns the result to variable X:

```
addr1='locate';
x=symexist(addr1);
```

character-expression
specifies a character expression that constructs a macro variable name without the leading ampersand.

Example
This example checks for the existence of a macro variable whose name is constructed from the current value of the DATA step variable StNum and assigns the result to variable X:

```
x=symexist('state'||left(stnum));
```

Details

The SYMEXIST function searches any enclosing local symbol tables and then the global symbol table for the indicated macro variable. The SYMEXIST function returns one of the following values:

- `1` if the macro variable is found
- `0` if the macro variable is not found

Example: Using SYMEXIST Function

The following example of the %TEST macro contains the SYMEXIST function:

```
%global x;
%macro test;
   %local y;
   data null;
      if symexist("x") then put "x EXISTS";
      else put "x does not EXIST";

      if symexist("y") then put "y EXISTS";
      else put "y does not EXIST";

      if symexist("z") then put "z EXISTS";
      else put "z does not EXIST";
   run;
%mend test;

%test;
```

In the previous example, executing the %TEST macro, which contains the SYMEXIST function, writes the following output to the SAS log:

```
x EXISTS
y EXISTS
z does not EXIST
```

SYMGET Function

Returns the value of a macro variable to the DATA step during DATA step execution.

Type:	DATA step function
Restriction:	The SYMGET function is not supported by the CAS engine.
See:	"RESOLVE Function" on page 299 "SYMGETN Function" on page 307 "CALL SYMPUT Routine" on page 289 "CALL SYMPUTN Routine" on page 294

Syntax

SYMGET(*argument*)

Required Argument

argument

specifies the name of a macro variable. *Argument* can be one of the following:

'macro-variable-name'

specifies within quotation marks the name of a macro variable without the leading ampersand.

When a macro variable value contains another macro variable reference, SYMGET does not attempt to resolve the reference. If *macro-variable-name* references a nonexistent macro variable, SYMGET returns a missing value.

Example This example assigns the value of macro variable G to the DATA step variable X.

```
x=symget('g');
```

character-variable-name

specifies the name of a DATA step character variable that contains the name of a macro variable without the leading ampersand.

If the value is not a valid SAS name, or if the macro processor cannot find a macro variable of that name, SAS writes a note to the log that the function has an invalid argument and sets the resulting value to missing.

Example These statements assign the value stored in the DATA step variable Code, which contains a macro variable name, to the DATA step variable Key:

```
length key $ 8;
input code $;
key=symget(code);
```

Each time the DATA step iterates, the value of variable Code supplies the name of a macro variable whose value is then assigned to variable Key.

character-expression

specifies a character expression that constructs a macro variable name without an ampersand.

Example This statement assigns to variable Score the value of the macro variable whose name is constructed using the letter s and the number of the current DATA step iteration (variable _N_):

```
score=symget('s'||left(_n_));
```

Details

SYMGET returns a character value that is the maximum length of a DATA step character variable. A returned value that is longer is truncated.

If SYMGET cannot locate the macro variable identified as the argument, it returns a missing value, and the program issues a message for an invalid argument to a function.

SYMGET can be used in all SAS language programs, including SCL programs. Because it resolves variables at program execution instead of macro execution, SYMGET should be used to return macro values to DATA step views, SQL views, and SCL programs.

Comparisons

- SYMGET returns values of macro variables during program execution, whereas the SYMPUT function assigns values that are produced by a program to macro variables during program execution.
- SYMGET accepts fewer types of arguments than the RESOLVE function. SYMGET resolves only a single macro variable. Using RESOLVE might result in the execution of macros and further resolution of values.
- SYMGET is available in all SAS programs, but SYMGETN is available only in SCL programs.

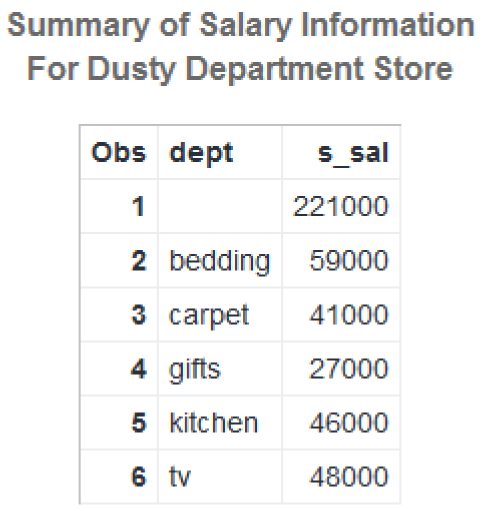
Example: Retrieving Variable Values Previously Assigned from a Data Set

```
data dusty;
   input dept $ name $ salary @@;
   datalines;
bedding Watlee 18000    bedding Ives 16000
bedding Parker 9000     bedding George 8000
bedding Joiner 8000     carpet Keller 20000
carpet Ray 12000        carpet Jones 9000
gifts Johnston 8000     gifts Matthew 19000
kitchen White 8000      kitchen Banks 14000
kitchen Marks 9000      kitchen Cannon 15000
tv Jones 9000           tv Smith 8000
tv Rogers 15000         tv Morse 16000
```

```
;
proc means noprint;
   class dept;
   var salary;
   output out=stats sum=s_sal;
run;
proc print data=stats;
   var dept s_sal;
   title "Summary of Salary Information";
   title2 "For Dusty Department Store";
run;
data _null_;
   set stats;
   if _n_=1 then call symput('s_tot',s_sal);
   else call symput('s'||dept,s_sal);
run;
data new;
   set dusty;
   pctdept=(salary/symget('s'||dept))*100;
   pcttot=(salary/&s_tot)*100;
run;
proc print data=new split="*";
   label dept   ="Department"
         name   ="Employee"
         pctdept="Percent of *Department* Salary"
         pcttot ="Percent of *   Store  * Salary";
   format pctdept pcttot 4.1;
   title  "Salary Profiles for Employees";
   title2 "of Dusty Department Store";
run;
```

This program produces the following output:

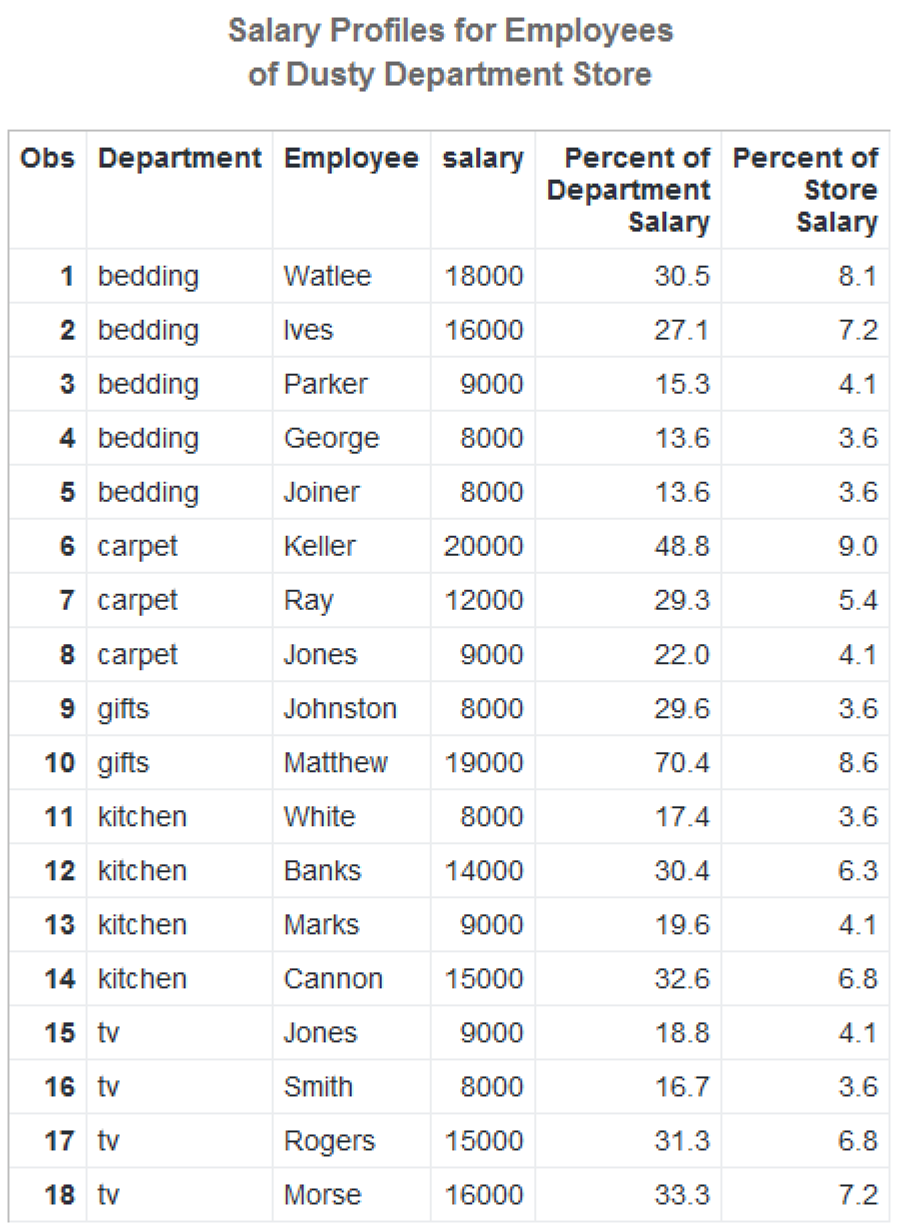
Output 16.1 Summary of Salary Information



Summary of Salary Information
For Dusty Department Store

Obs	dept	s_sal
1		221000
2	bedding	59000
3	carpet	41000
4	gifts	27000
5	kitchen	46000
6	tv	48000

Output 16.2 *Salary Profiles for Employees*



Salary Profiles for Employees
of Dusty Department Store

Obs	Department	Employee	salary	Percent of Department Salary	Percent of Store Salary
1	bedding	Watlee	18000	30.5	8.1
2	bedding	Ives	16000	27.1	7.2
3	bedding	Parker	9000	15.3	4.1
4	bedding	George	8000	13.6	3.6
5	bedding	Joiner	8000	13.6	3.6
6	carpet	Keller	20000	48.8	9.0
7	carpet	Ray	12000	29.3	5.4
8	carpet	Jones	9000	22.0	4.1
9	gifts	Johnston	8000	29.6	3.6
10	gifts	Matthew	19000	70.4	8.6
11	kitchen	White	8000	17.4	3.6
12	kitchen	Banks	14000	30.4	6.3
13	kitchen	Marks	9000	19.6	4.1
14	kitchen	Cannon	15000	32.6	6.8
15	tv	Jones	9000	18.8	4.1
16	tv	Smith	8000	16.7	3.6
17	tv	Rogers	15000	31.3	6.8
18	tv	Morse	16000	33.3	7.2

SYMGETN Function

In SAS Component Control Language (SCL) programs, returns the value of a global macro variable as a numeric value when the SCL program executes.

Type:	SCL function
Restriction:	The SYMPUT CALL routine is not supported by the CAS engine.
Notes:	It is inefficient to use SYMGETN to retrieve values that are not assigned with SYMPUTN and values that are not numeric.

You can also use SYMGETN to retrieve the value of a macro variable whose name is stored in an SCL variable.

See: "SYMGET Function" on page 303
"CALL SYMPUT Routine" on page 289
"CALL SYMPUTN Routine" on page 294

Syntax

SCL-variable-name=**SYMGETN**(*argument*)

Required Argument

argument

specifies the name of a global macro variable. *Argument* can be one of the following:

'global-macro-variable-name'

specifies within single quotation marks the name of a global macro variable without the leading ampersand.

Note: If SYMGETN cannot locate *macro-variable-name*, it returns a missing value.

Tip: To return the value stored in a macro variable when an SCL program compiles, use a macro variable reference in an assignment statement:

```
SCL variable=&macro-variable;
```

Example: This example stores the value of the macro variable Unit in the SCL variable UnitNum when the SCL program executes:

```
unitnum=symgetn('unit');
```

SCL-variable-name

specifies the name of the name of an SCL variable that contains the name of a global macro variable.

Example: This example retrieves the value of SCL variable UnitVar, whose value is 'UNIT',and assigns the value to SCL variable UnitNum:

```
unitnum=symgetn(unitvar)
```

Comparisons

- SYMGETN is available only in SCL programs, but SYMGET is available in DATA step programs and SCL programs.
- SYMGETN retrieves values, but SYMPUTN assigns values.

SYMGLOBL Function

Returns an indication as to whether a macro variable is global in scope to the DATA step during DATA step execution.

Type:	DATA step function

Syntax

SYMGLOBL(*argument*)

Required Argument

argument

specifies the name of a macro variable. *Argument* can be one of the following:

'*macro-variable-name*'

specifies within quotation marks the name of a macro variable without the leading ampersand.

character-variable-name

specifies the name of a DATA step character variable that contains a macro variable name.

character-expression

specifies a character expression that constructs a macro variable name.

Details

The SYMGLOBL function searches enclosing scopes for the indicated macro variable and returns a value of `1` if the macro variable is found in the global symbol table, otherwise it returns a `0`. For more information about the global and local symbol tables and macro variable scopes, see “Scopes of Macro Variables” on page 57.

Example: Using SYMGLOBL Function

The following example of the %TEST macro contains the SYMGLOBL function:

```
%global x;
%macro test;
   %local y;
   data null;
      if symglobl("x") then put "x is GLOBAL";
```

```
         else put "x is not GLOBAL";

         if symglobl("y") then put "y is GLOBAL";
         else put "y is not GLOBAL";

         if symglobl("z") then put "z is GLOBAL";
         else put "z is not GLOBAL";
      run;
   %mend test;

   %test;
```

In the previous example, executing the %TEST macro, which contains the SYMGLOBL function, writes the following output to the SAS log:

```
x is GLOBAL
y is not GLOBAL
z is not GLOBAL
```

SYMLOCAL Function

Returns an indication as to whether a macro variable is local in scope to the DATA step during DATA step execution.

Type: DATA step function

Syntax

SYMLOCAL(*argument*)

Required Argument

argument
specifies the name of a macro variable. *Argument* can be one of the following:

'macro-variable-name'
specifies within quotation marks the name of a macro variable without the leading ampersand.

character-variable-name
specifies without quotation marks the name of a DATA step character variable that contains a macro variable name.

character-expression
specifies a character expression that constructs a macro variable name.

Details

The SYMLOCAL function searches enclosing scopes for the indicated macro variable and returns a value of `1` if the macro variable is found in a local symbol table, otherwise it returns a `0`. For more information about the global and local symbol tables and macro variable scopes, see “Scopes of Macro Variables” on page 57.

Example: Using SYMLOCAL Function

The following example of the %TEST macro contains the SYMLOCAL function:

```
%global x;
%macro test;
   %local y;
   data null;
      if symlocal("x") then put "x is LOCAL";
      else put "x is not LOCAL";

      if symlocal("y") then put "y is LOCAL";
      else put "y is not LOCAL";

      if symlocal("z") then put "z is LOCAL";
      else put "z is not LOCAL";
   run;
%mend test;

%test;
```

In the previous example, executing the %TEST macro, which contains the SYMLOCAL function, writes the following output to the SAS log:

```
x is not LOCAL
y is LOCAL
z is not LOCAL
```

17

Macro Functions

Macro Functions

A macro language function processes one or more arguments and produces a result.

Dictionary

%BQUOTE Macro Function

Mask special characters and mnemonic operators in a resolved value at macro execution.

Type:	Macro quoting function
Note:	The maximum level of nesting for the macro quoting functions is 10.
Tip:	You can use the %BQUOTE function for all execution-time macro quoting because it masks all characters and mnemonic operators that can be interpreted as elements of macro language.
See:	"%QUOTE Macro Function" on page 334 "%SUPERQ Macro Function" on page 347

Syntax

%BQUOTE(*character-string*)

Required Argument

character-string
specifies a character string or an expression that results in a character string that contains one or more special characters and mnemonic operators. The following special characters and mnemonic operators in the specified string are masked at execution time:

```
' " ( ) + - * / < > = ¬ ^ ~ ; , # blank
AND OR NOT EQ NE LE LT GE GT IN
```

Details

The %BQUOTE function masks a character string or resolved value of a text expression during execution of a macro or macro language statement. It masks the following special characters and mnemonic operators:

```
' " ( ) + - * / < > = ¬ ^ ~ ; , # blank
AND OR NOT EQ NE LE LT GE GT IN
```

Quotation marks (`'"`) do not have to be marked.

For a description of quoting in SAS macro language, see "Macro Quoting" on page 96.

Comparisons

%BQUOTE does not require that you mark quotation marks.

%BQUOTE does not mask characters `&` and `%` while %NRBQUOTE does.

Example: Quoting a Variable

This example tests whether a filename passed to the macro FILEIT starts with a quotation mark. Based on that evaluation, the macro creates the correct FILE command.

```
%macro fileit(infile);
  %if %bquote(&infile) NE %then
     %do;
         %let char1 = %bquote(%substr(&infile,1,1));
         %if %bquote(&char1) = %str(%')
             or %bquote(&char1) = %str(%")
         %then %let command=FILE &infile;
         %else %let command=FILE "&infile";
     %end;
  %put &command;
%mend fileit;
%fileit(myfile)
%fileit('myfile')
```

When this program executes, the following is written to the log:

```
FILE "myfile"
FILE 'myfile'
```

%EVAL Macro Function

Evaluates arithmetic and logical expressions using integer arithmetic.

Type:	Macro evaluation function
See:	"%SYSEVALF Macro Function" on page 352

Syntax

%EVAL(*expression*)

Required Argument

expression

specifies an integer arithmetic expression or a logical expression to evaluate. For an arithmetic expression, any fraction in the result is truncated. For a logical expression, 1 is returned if the expression resolves to True. Otherwise, 0 is returned.

Examples This example adds 10 and 5 and assigns the result to macro variable X:

```
%let x=%eval(10+5);
```

This example evaluates whether the value of macro variable A is greater than or equal to 10 and assigns the result to macro variable X:

```
%let x=%eval(&a ge 10);
```

Details

The %EVAL function evaluates integer arithmetic or logical expressions. %EVAL operates by converting its argument from a character value to a numeric or logical expression. Then, it performs the evaluation. Finally, %EVAL converts the result back to a character value and returns that value.

If all operands can be interpreted as integers, the expression is treated as arithmetic. If at least one operand cannot be interpreted as numeric, the expression is treated as logical. If a division operation results in a fraction, the fraction is truncated to an integer.

Logical, or Boolean, expressions return a value that is evaluated as true or false. In the macro language, any numeric value other than 0 is true and a value of 0 is false.

%EVAL accepts only operands in arithmetic expressions that represent integers (in standard or hexadecimal form). Operands that contain a period character cause an error when they are part of an integer arithmetic expression. The following examples show correct and incorrect usage, respectively:

```
%let d=%eval(10+20);       /* Correct usage   */
%let d=%eval(10.0+20.0);   /* Incorrect usage */
```

Because %EVAL does not convert a value containing a period to a number, the operands are evaluated as character operands. When %EVAL encounters a value containing a period, it displays an error message about finding a character operand where a numeric operand is required.

An expression that compares character values in the %EVAL function uses the sort sequence of the operating environment for the comparison. For more information about operating environment sort sequences, see "SORT Procedure" in *Base SAS Procedures Guide*.

Note: All parts of the macro language that evaluate expressions (for example, %IF and %DO statements) call %EVAL to evaluate the condition. For a complete discussion of how macro expressions are evaluated, see Chapter 6, "Macro Expressions," on page 85.

Comparisons

%EVAL performs integer evaluations, but %SYSEVALF performs floating point evaluations.

Examples

Example 1: Illustrating Integer Arithmetic Evaluation

These statements illustrate different types of evaluations:

```
%let a=1+2;
%let b=10*3;
%let c=5/3;
%let eval_a=%eval(&a);
%let eval_b=%eval(&b);
%let eval_c=%eval(&c);
%put &a is &eval_a;
%put &b is &eval_b;
%put &c is &eval_c;
```

When these statements are submitted, the following is written to the SAS log:

```
1+2 is 3
10*3 is 30
5/3 is 1
```

The third %PUT statement shows that %EVAL discards the fractional part when it performs division on integers that would result in a fraction:

Example 2: Incrementing a Counter

The macro TEST uses %EVAL to increment the value of the macro variable I by 1. Also, the %DO %WHILE statement calls %EVAL to evaluate whether I is greater than the value of the macro variable FINISH.

```
%macro test(finish);
   %let i=1;
   %do %while (&i<&finish);
      %put the value of i is &i;
      %let i=%eval(&i+1);
   %end;
%mend test;
%test(5)
```

When this program executes, these lines are written to the SAS log:

```
The value of i is 1
The value of i is 2
The value of i is 3
The value of i is 4
```

%INDEX Macro Function

Returns the position of the first character of a string.

Type: Macro function

Syntax

%INDEX(*source-string*, *target-string*)

Required Arguments

source-string
specifies a character string or a text expression that results in a character string as the source string.

target-string
specifies a character string or a text expression that results in a character string as the target string.

Details

The %INDEX function searches *source-string* for the first occurrence of *target-string* and returns the position of its first character. If *target-string* is not found, the function returns 0.

Example: Locating a Character

The following statements find the first character `V` in a string:

```
%let a=a very long value;
%let b=%index(&a,v);
%put V appears at position &b;
```

When these statements execute, the following line is written to the SAS log:

```
V appears at position 3.
```

%LENGTH Macro Function

Returns the length of a string.

Type:	Macro function

Syntax

%LENGTH(*character-string*)

Required Argument

character-string
specifies a character string or a text expression that results in a character string.

Details

If the argument is a character string, %LENGTH returns the length of the string. If the argument is a text expression, %LENGTH returns the length of the resolved value. If the argument has a null value, %LENGTH returns 0.

Comparisons

Macro function %LENGTH differs from SAS function LENGTH in the following ways:

- Macro function %LENGTH does not count leading and trailing blanks unless a macro quoting function is used on the macro variable. The SAS LENGTH function counts leading blanks, but not trailing blanks.
- Macro function %LENGTH counts bytes, not characters. SAS function LENGTH counts bytes for fixed-length strings or counts characters for VARCHAR strings.

Example: Returning String Lengths

The following statements find the lengths of character strings and text expressions.

```
%let a=Happy;
%let b=Birthday;
%put The length of &a is %length(&a).;
%put The length of &b is %length(&b).;
%put The length of &a &b To You is %length(&a &b to you).;
```

When these statements execute, the following is written to the SAS log:

```
The length of Happy is 5.
The length of Birthday is 8.
The length of Happy Birthday To You is 21.
```

%NRBQUOTE Macro Function

Masks special characters, including & and %, and mnemonic operators in a resolved value at macro execution.

Type:	Macro quoting function
Note:	The maximum level of nesting for the macro quoting functions is 10.
Tip:	You can use the %NRBQUOTE function for all execution-time macro quoting because it masks all characters and mnemonic operators that can be interpreted as elements of macro language.
See:	"%BQUOTE Macro Function" on page 314

Syntax

%NRBQUOTE(*character-string*)

Required Argument

character-string

specifies a character string or a text expression that results in a character string that might contain one or more special characters and mnemonic operators. The following special characters and mnemonic operators in the specified string are masked at execution time:

```
& % ' " ( ) + - * / < > = ¬ ^ ~ ; , # blank
AND OR NOT EQ NE LE LT GE GT IN
```

Details

The %NRBQUOTE function masks a character string or resolved value of a text expression during execution of a macro or macro language statement. It masks the following special characters and mnemonic operators:

```
& % ' " ( ) + − * / < > = ¬ ^ ~ ; , # blank
AND OR NOT EQ NE LE LT GE GT IN
```

%NRBQUOTE is most useful when the resolved value of an argument might contain:

- strings that look like macro variable references but are not, so the macro processor should not attempt to resolve them when it next encounters them.
- macro invocations that you do not want the macro processor to attempt to resolve when it next encounters them.

Quotation marks (`'"`) do not have to be marked.

For a description of quoting in SAS macro language, see "Macro Quoting" on page 96.

Comparisons

%NRBQUOTE and the %SUPERQ function mask the same items. However, %SUPERQ does not attempt to resolve a macro variable reference or a macro invocation that occurs in the value of the specified macro variable. %NRBQUOTE does attempt to resolve such references. %NRBQUOTE does not require that you mark quotation marks.

%NRBQUOTE masks characters `&` and `%` while %BQUOTE does not.

See Also

"%BQUOTE Macro Function" on page 314

%NRQUOTE Macro Function

Masks special characters, including & and %, and mnemonic operators in a resolved value at macro execution.

Type:	Macro quoting function
Note:	The maximum level of nesting for the macro quoting functions is 10.
See:	"%QUOTE Macro Function" on page 334

Syntax

%NRQUOTE(*character-string*)

Required Argument

character-string

specifies a character string or an expression that results in a character string that contains one or more special characters and mnemonic operators. The following special characters and mnemonic operators in the specified string are masked at execution time:

```
& % ' " ( ) + – * / < > = ¬ ^ ~ ; , # blank
AND OR NOT EQ NE LE LT GE GT IN
```

Details

The %NRQUOTE function masks a character string or resolved value of a text expression during execution of a macro or macro language statement. It masks the following special characters and mnemonic operators:

```
& % + – * / < > = ¬ ^ ~ ; , # blank
AND OR NOT EQ NE LE LT GE GT IN
```

It also masks the following characters when they occur in pairs and when they are not matched and are marked by a preceding `%`:

```
' "
```

%NRQUOTE is most useful when an argument might contain a macro variable reference or macro invocation that you do not want resolved.

For a description of quoting in SAS macro language, see “Macro Quoting” on page 96.

Comparisons

- %NRQUOTE masks the same items as %STR and %NRSTR, respectively. However, %STR and %NRSTR mask constant text instead of a resolved value. And, %STR and %NRSTR work when a macro compiles, and %NRQUOTE work when a macro executes.
- The %NRBQUOTE function does not require that quotation marks without a match be marked with a preceding `%` while %NRQUOTE does.
- %NRQUOTE masks resolved values, and the %SUPERQ function prevents resolution of any macro invocations or macro variable references that might occur in a value.
- %NRQUOTE masks `&` and `%` while %QUOTE does not.

%NRSTR Macro Function

Masks special characters, including & and %, and mnemonic operators in constant text during macro compilation.

Type:	Macro quoting function
Note:	The maximum level of nesting for the macro quoting functions is 10.
See:	"%STR Macro Function" on page 342

Syntax

%NRSTR(*character-string*)

Required Argument

character-string

specifies a character string or an expression that results in a character string that contains one or more special characters and mnemonic operators. The following special characters and mnemonic operators in the specified string are masked at macro compilation time:

```
& % + - * / < > = ¬ ^ ~ ; ,  # blank
AND OR NOT EQ NE LE LT GE GT IN
```

Details

The %NRSTR function masks a character string during compilation of a macro or macro language statement. It masks the following special characters and mnemonic operators:

```
& % + - * / < > = ¬ ^ ~ ; ,  # blank
AND OR NOT EQ NE LE LT GE GT IN
```

It also masks the following characters when they occur in pairs and when they are not matched and are marked by a preceding `%`:

```
' " ( )
```

Table 17.1 *Using %STR and %NRSTR Arguments*

Argument	Use
Percent sign before a quotation mark - for example, %' or %",	Percent sign with quotation mark

Argument	Use
	EXAMPLE: %let percent=%str(Jim%'s office);

Percent sign before a parenthesis - for example, %(or %)	Two percent signs (%%):
	EXAMPLE: %let x=%str(20%%);

Character string with the comment symbols /* or -->	%STR with each character EXAMPLE: %str(/) %str(*) *comment-text* %str(*)%str(/)

%NRSTR is most useful for character strings that contain:

- a semicolon that should be treated as text rather than as part of a macro program statement
- blanks that are significant
- a quotation mark or parenthesis without a match
- one or more characters that require masking such as special characters and mnemonic operators.

Putting the same argument within nested %NRSTR and %NRQUOTE functions is redundant. This example shows an argument that is masked at macro compilation by the %NRSTR function and remains masked at macro execution. Thus, in this example, the %NRQUOTE function used here has no effect.

```
%nrquote(%nrstr(argument))
```

CAUTION
Do not use %NRSTR to enclose other macro functions or macro invocations that have a list of parameter values. Because %NRSTR masks parentheses without a match, the macro processor does not recognize the arguments of a function or the parameter values of a macro invocation.

For a description of quoting in SAS macro language, see "Macro Quoting" on page 96.

%QSCAN Macro Function

Searches for a word and masks special characters and mnemonic operators.

Type:	Macro function

Syntax

%QSCAN(*character-string*, *n*<, *charlist*<, *modifiers*>>)

Required Arguments

character-string

specifies a character string or a text expression that results in a character string. If the specified string contains a comma, enclose *character-string* in a quoting function such as %BQUOTE(*character-string*).

n

specifies an integer or a text expression that yields an integer, which specifies the position of the word to return. An implied %EVAL gives *n* numeric properties. If *n* is greater than the number of words in *argument*, the functions return a null string.

Note: When you are using Version 8 or greater, if *n* is negative, %QSCAN examines the character string and selects the word that starts at the end of the string and searches backward.

Optional Arguments

charlist

specifies an optional character expression that initializes a list of characters. This list determines which characters are used as the delimiters that separate words. The following rules apply:

- By default, all characters in *charlist* are used as delimiters.
- If you specify the K modifier in the *modifier* argument, then all characters that are not in *charlist* are used as delimiters.

Tip You can add more characters to *charlist* by using other modifiers.

modifiers

specifies a character constant, a variable, or an expression in which each non-blank character modifies the action of the %QSCAN function. Blanks are ignored. You can use the following characters as modifiers:

a or A	adds alphabetic characters to the list of characters.
b or B	scans backward from right to left instead of from left to right, regardless of the sign of the *count* argument.
c or C	adds control characters to the list of characters.
d or D	adds digits to the list of characters.
f or F	adds an underscore and English letters (that is, valid first characters in a SAS variable name using VALIDVARNAME=V7) to the list of characters.
g or G	adds graphic characters to the list of characters. Graphic characters are characters that, when printed, produce an image on paper.
h or H	adds a horizontal tab to the list of characters.
i or I	ignores the case of the characters.

k or K	causes all characters that are not in the list of characters to be treated as delimiters. That is, if K is specified, then characters that are in the list of characters are kept in the returned value rather than being omitted because they are delimiters. If K is not specified, then all characters that are in the list of characters are treated as delimiters.
l or L	adds lowercase letters to the list of characters.
m or M	specifies that multiple consecutive delimiters, and delimiters at the beginning or end of the *string* argument, refer to words that have a length of zero. If the M modifier is not specified, then multiple consecutive delimiters are treated as one delimiter, and delimiters at the beginning or end of the *string* argument are ignored.
n or N	adds digits, an underscore, and English letters (that is, the characters that can appear in a SAS variable name using VALIDVARNAME=V7) to the list of characters.
o or O	processes the *charlist* and *modifier* arguments only once, rather than every time the %QSCAN function is called. Using the O modifier in the DATA step (excluding WHERE clauses), or in the SQL procedure can make %QSCAN run faster when you call it in a loop where the *charlist* and *modifier* arguments do not change. The O modifier applies separately to each instance of the %QSCAN function in your SAS code, and does not cause all instances of the %QSCAN function to use the same delimiters and modifiers.
p or P	adds punctuation marks to the list of characters.
q or Q	ignores delimiters that are inside substrings that are enclosed in quotation marks. If the value of the *string* argument contains unmatched quotation marks, then scanning from left to right will produce different words than scanning from right to left.
r or R	removes leading and trailing blanks from the word that %QSCAN returns. If you specify both the Q and R modifiers, then the %QSCAN function first removes leading and trailing blanks from the word. Then, if the word begins with a quotation mark, %QSCAN also removes one layer of quotation marks from the word.
s or S	adds space characters to the list of characters (blank, horizontal tab, vertical tab, carriage return, line feed, and form feed).
t or T	trims trailing blanks from the *string* and *charlist* arguments. If you want to remove trailing blanks from only one character argument instead of both character arguments, then use the TRIM function instead of the %QSCAN function with the T modifier.
u or U	adds uppercase letters to the list of characters.
w or W	adds printable (writable) characters to the list of characters.
x or X	adds hexadecimal characters to the list of characters.

Tip If the *modifier* argument is a character constant, then enclose it in quotation marks. Specify multiple modifiers in a single set of quotation marks. A *modifier* argument can also be expressed as a character variable or expression.

See Also

"%SCAN Macro Function" on page 337

%QSUBSTR Macro Function

Produces a substring and masks special characters and mnemonic operators.

Type: Macro function

Syntax

%QSUBSTR(*character-string*, *position*<, *length*>)

Required Arguments

character-string

specifies a character string or an expression that results in a character string that contains special characters and mnemonic operators. The following special characters and mnemonic operators in the specified string are masked:

```
& % ' " ( ) + - * / < > = ¬ ^ ~ ; , #  blank
AND OR NOT EQ NE LE LT GE GT IN
```

If the specified string does not contain any of these characters, you can use %SUBSTR instead. See "%SUBSTR Macro Function" on page 345.

position

specifies an integer or an expression (text, logical, or arithmetic) that yields an integer, which specifies the position of the first character in the substring. If *position* is greater than the number of characters in the string, %QSUBSTR issues a warning message and return a null value. An automatic call to %EVAL causes *n* to be treated as a numeric value.

Optional Argument

length

specifies an optional integer or an expression (text, logical, or arithmetic) that yields an integer that specifies the number of characters in the substring. If *length* is greater than the number of characters following *position* in *argument*, %QSUBSTR issues a warning message and return a substring containing the characters from *position* to the end of the string. By default, %QSUBSTR produces a string containing the characters from *position* to the end of the character string.

Details

The %QSUBSTR function produces a substring of *argument*, beginning at *position*, for *length* number of characters.

%QSUBSTR masks the following special characters and mnemonic operators:

```
& % ' " ( ) + - * / < > = ¬ ^ ~ ; , #  blank
AND OR NOT EQ NE LE LT GE GT IN
```

Comparisons

%QSUBSTR masks special characters and mnemonic operators. %SUBSTR does not mask these characters.

%QSUBSTR masks the same characters as the %NRBQUOTE function.

Example: Storing a Long Macro Variable Value in Segments

The macro SEPMSG separates the value of the macro variable MSG into 40-character units and stores each unit in a separate variable.

```
%macro sepmsg(msg);
   %let i=1;
   %let start=1;
   %if %length(&msg)>40 %then
      %do;
          %do %until(%length(&&msg&i)<40);
             %let msg&i=%qsubstr(&msg,&start,40);
             %put Message &i is: &&msg&i;
             %let i=%eval(&i+1);
             %let start=%eval(&start+40);
             %let msg&i=%qsubstr(&msg,&start);
          %end;
          %put Message &i is: &&msg&i;
      %end;
   %else %put No subdivision was needed.;
%mend sepmsg;
%sepmsg(%nrstr(A character operand was found in the %EVAL function
or %IF condition where a numeric operand is required.  A character
operand was found in the %EVAL function or %IF condition where a
numeric operand is required.));
```

When this program executes, these lines are written to the SAS log:

```
Message 1 is: A character operand was found in the %EV
Message 2 is: AL function or  %IF condition where a nu
Message 3 is: meric operand is required.  A character
Message 4 is: operand was  found in the %EVAL function
Message 5 is:  or %IF condition where a numeric operan
Message 6 is: d is required.
```

See Also

"Example 2: Comparing Actions of %SUBSTR and %QSUBSTR" on page 346

%QSYSFUNC Macro Function

Executes functions and masks special characters and mnemonic operators.

Type: Macro function

Syntax

%QSYSFUNC(*function*(*arguments*)<, *format*>)

Required Argument

function(arguments)

specifies the name of a function to execute and its arguments.

function

specifies the name of the function to execute.

This function can be a SAS function, a function written with SAS/TOOLKIT software, or a function created using the "FCMP Procedure" in *Base SAS Procedures Guide*. The function cannot be a macro function.

All SAS functions, except those listed in Table 17.49 on page 332, can be used with %QSYSFUNC.

You cannot nest functions to be used with a single %QSYSFUNC. However, you can nest %QSYSFUNC calls:

```
%let x=%qsysfunc(trim(%qsysfunc(left(&num))));
```

Syntax for Selected Functions Used with the %QSYSFUNC Function on page 531 shows the syntax of SAS functions used with %QSYSFUNC.

arguments

specifies one or more arguments used by *function*.

An argument can be a macro variable reference or a text expression that produces arguments for a function. %QSYSFUNC masks the following special characters and mnemonic operators that appear in *argument(s)*:

```
& % ' " ( ) + - * / < > = ¬ ^ ~ ; , #  blank
AND OR NOT EQ NE LE LT GE GT IN
```

Optional Argument

format

specifies an optional format to apply to the result of *function*. This format can be provided by SAS, generated by PROC FORMAT, or created with SAS/TOOLKIT. There is no default value for *format*. If you do not specify a *format*, the SAS macro facility does not perform a *format* operation on the result and uses the default of the *function*.

Details

Specifying Function Arguments

Specifying Character Argument Values

Because %QSYSFUNC is a macro function, you do not need to enclose character values in quotation marks as you do in DATA step 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 %QSYSFUNC. These statements show the difference:

- function OPEN used in DATA step statements:

  ```
  dsid=open("Sasuser.Houses","i");
  dsid=open("&mydata","&mode");
  ```

- function OPEN used within %QSYSFUNC:

  ```
  %let dsid = %qsysfunc(open(Sasuser.Houses,i));
  %let dsid=%qsysfunc(open(&mydata,&mode));
  ```

All arguments in DATA step functions within %QSYSFUNC must be separated by commas. You cannot use argument lists preceded by the word OF.

Note: The arguments to %QSYSFUNC are evaluated according to the rules of the SAS macro language. This includes both the function name and the argument list to the function. In particular, an empty argument position will not generate a NULL argument, but a zero-length argument.

Specifying Numeric Argument Values

When a function called by %QSYSFUNC requires a numeric argument, %QSYSFUNC converts the argument to a numeric value. The value can be a number, an expression that evaluates to a number, or a function that returns a number. Here is an example.

```
%put Result: %qsysfunc(mean(83, 6 * 2, %qsysfunc(divide(50,2))));
```

```
Result: 40
```

Specifying Argument Values That Can Be Character or Numeric

Some SAS functions such as SUBSTRN and CATX accept a numeric or character value as one or more arguments. For these arguments, %QSYSFUNC first silently attempts to convert the argument value to a numeric value using the %SYSEVALF on page 352 function. If the %SYSEVALF function fails, %QSYSFUNC passes the argument value as-is to the called function. Here is a simple example.

```
%put %qsysfunc(catx(%str( ), Result:, (1 + (5 * 8)) / 4));
```

The first argument in the CATX function is a constant string, so %QSYSFUNC passes the value (blank) to the CATX function. The subsequent CATX function arguments can be numeric or character. In this example, %QSYSFUNC first attempts to silently convert the second argument value to a numeric value using the %SYSEVALF function. Because `%SYSEVALF(Result:)` fails, %QSYSFUNC passes the value `Result:` to the CATX function as the second argument value. When %QSYSFUNC attempts to convert the third argument to a numeric value, the %SYSEVALF function returns 10.25. %QSYSFUNC then passes 10.25 to the CATX function as the third argument value. Here is the result:

```
Result: 10.25
```

In some cases, this behavior can cause unexpected results. Consider the following example.

```
%put %qsysfunc(catx(%str( ), Wind:, NE, 15-25 MPH));
```

```
Wind: 0 15-25 MPH
```

In this case, when %QSYSFUNC evaluates the second argument value, %SYSEVALF(NE) returns 0 because NE is the not-equals mnemonic operator, and null NE null is False. %QSYSFUNC passes 0 to the CATX function as the second argument value instead of NE. Similar issues can happen when argument values contain mathematical symbols.

To prevent NE from being evaluated to 0 in this example, use nested %NRSTR on page 323 and %STR on page 342 functions as follows:

```
%put %qsysfunc(catx(%str( ), Wind:, %nrstr(%str(NE)), 15-25 MPH));
```

The %NRSTR function masks the %STR function result in such a way that the %STR function result is evaluated inside the CATX function. This ensures that the literal NE is passed in as the second argument instead of 0. Here is the result.

```
Wind: NE 15-25 MPH
```

If an argument cannot be evaluated by %SYSEVALF, an error results, but execution continues and the argument is passed to the called function. This can occur if an argument is one or more blank spaces or is a null value. In either case, there is no expression for %SYSEVALF to evaluate. Here is a simple example:

```
%put %qsysfunc(cat(A, %str( ), B));
```

Because the second argument is a single space, %SYSEVALF has nothing to evaluate, so it writes the following message to the SAS log:

```
ERROR: %SYSEVALF function has no expression to evaluate.
```

However, execution continues and the space is passed to the CAT function, which then correctly concatenates the strings:

```
A B
```

In this case, to eliminate the error, use the CATX function instead and specify a blank space as the delimiter as shown in the following example:

```
%put %qsysfunc(catx(%str( ), A, B));
```

The result is the same.

SAS Functions That Are Not Supported

The following table lists the SAS functions that are not available with macro function %QSYSFUNC.

Table 17.2 SAS Functions Not Available with %SYSFUNC and %QSYSFUNC

All Variable Information Functions	ALLCOMB	ALLPERM
DIF	DIM	HBOUND
INPUT	IORCMSG	LAG
LBOUND	LEXCOMB	LEXCOMBI
LEXPERK	LEXPERM	METADATA_GETNOBJ
MISSING	PUT	RESOLVE
SYMGET		
All SAS File I/O Functions	MD5	SHA256

Note: INPUT and PUT are not available with %QSYSFUNC. Use INPUTN, INPUTC, PUTN, and PUTC instead.

Note: The Variable Information functions include functions such as VNAME and VLABEL. For a complete list, see “Definitions of Functions and CALL Routines” in *SAS Functions and CALL Routines: Reference*.

CAUTION
Values returned by SAS functions might be truncated. Although values returned by macro functions are not limited to the length imposed by the DATA step, values returned by SAS functions do have that limitation.

Function Result

%QSYSFUNC masks the following special characters and mnemonic operators in its result:

```
& % ' " ( ) + − * / < > = ¬ ^ ~ ; , #  blank
```

```
AND OR NOT EQ NE LE LT GE GT IN
```

%QSYSFUNC can return a floating point number when the function that it executes supports floating point numbers.

Comparisons

%QSYSFUNC masks the same characters and mnemonic operators as the %NRBQUOTE on page 320 function. %SYSFUNC does not mask these characters and mnemonic operators.

Example: Masking Special Characters in a Function Result with %QSYSFUNC

This simple example demonstrates the difference between %SYSFUNC and %QSYSFUNC, and how to use %QSYSFUNC to mask special characters in the result of the CATX function. The following code uses %SYSFUNC and function CATX to concatenate strings Safe and Sound using & as a delimiter.

```
%put %sysfunc(catx(&, Safe, Sound));
```

When this statement is executed, the following warning is written to the SAS log:

```
WARNING: Apparent symbolic reference SOUND not resolved.
```

The result from the CATX function is Safe&Sound. %PUT attempts to resolve macro variable sound, but in this example, it does not exist. When macro variable sound does exist, unexpected results can occur:

```
%let sound = 40 dB;
%put %sysfunc(catx(&, Safe, Sound));
```

```
Safe40 dB
```

To mask & in the CATX function result, use %QSYSFUNC instead:

```
%put %qsysfunc(catx(&, Safe, Sound));
```

This produces the expected result:

```
Safe&Sound
```

See Also

"%SYSFUNC Macro Function" on page 356

%QUOTE Macro Function

Mask special characters and mnemonic operators in a resolved value at macro execution.

Type:	Macro quoting function
Note:	The maximum level of nesting for the macro quoting functions is 10.
See:	"%BQUOTE Macro Function" on page 314, "%NRBQUOTE Macro Function" on page 320, "%NRSTR Macro Function" on page 323, and "%SUPERQ Macro Function" on page 347

Syntax

%QUOTE(*character-string*)

Required Argument

character-string

specifies a character string or an expression that results in a character string that contains one or more special characters and mnemonic operators. The following special characters and mnemonic operators in the specified string are masked at execution time:

```
+ – * / < > = ¬ ^ ~ ; , # blank
AND OR NOT EQ NE LE LT GE GT IN
```

Details

The %QUOTE function masks a character string or resolved value of a text expression during execution of a macro or macro language statement. It masks the following special characters and mnemonic operators:

```
+ – * / < > = ¬ ^ ~ ; , # blank
AND OR NOT EQ NE LE LT GE GT IN
```

It also masks the following characters when they occur in pairs and when they are not matched and are marked by a preceding `%`

```
' "
```

For a description of quoting in SAS macro language, see "Macro Quoting" on page 96.

Comparisons

- %QUOTE masks the same items as %STR and %NRSTR, respectively. However, %STR and %NRSTR mask constant text instead of a resolved value. And, %STR and %NRSTR work when a macro compiles, and %QUOTE and %NRQUOTE work when a macro executes.
- The %BQUOTE and %NRBQUOTE functions do not require that quotation marks without a match be marked with a preceding `%`, and %QUOTE and %NRQUOTE do.
- %QUOTE and %NRQUOTE mask resolved values, and the %SUPERQ function prevents resolution of any macro invocations or macro variable references that might occur in a value.
- %QUOTE does not mask `&` and `%` while %NRQUOTE does.

Example: Quoting a Value That Might Contain a Mnemonic Operator

The macro DEPT1 receives abbreviations for states and therefore might receive the value OR for Oregon.

```
%macro dept1(state);
      /* without %quote -- problems might occur */
   %if &state=nc %then
       %put North Carolina Department of Revenue;
   %else %put Department of Revenue;
%mend dept1;
%dept1(or)
```

When the macro DEPT1 executes, the %IF condition executes a %EVAL function, which evaluates `or` as a logical operator in this expression. Then the macro processor produces an error message for an invalid operand in the expression `or=nc`.

The macro DEPT2 uses the %QUOTE function to treat characters that result from resolving &STATE as text:

```
%macro dept2(state);
      /* with %quote function--problems are prevented */
   %if %quote(&state)=nc %then
       %put North Carolina Department of Revenue;
   %else %put Department of Revenue;
%mend dept2;
%dept2(or)
```

The %IF condition now compares the strings `or` and `nc` and writes to the SAS log:

```
Department of Revenue
```

%QUPCASE Macro Function

Converts a value to uppercase and returns a result that masks special characters and mnemonic operators.

Type:	Macro function
Tip:	To convert characters to lowercase, use the %LOWCASE or %QLOWCASE autocall macro.

Syntax

%QUPCASE(*character-string*)

Required Argument

character-string

specifies a character string or an expression that results in a character string that contains one or more special characters and mnemonic operators. The following special characters and mnemonic operators in the specified string are masked at execution time:

```
& % ' " ( ) + − * / < > = ¬ ^ ~ ; , # blank
AND OR NOT EQ NE LE LT GE GT IN
```

Details

The %QUPCASE function converts lowercase characters in the argument to uppercase. %QUPCASE masks the following special characters and mnemonic operators in its result:

```
& % ' " ( ) + − * / < > = ¬ ^ ~ ; , # blank
AND OR NOT EQ NE LE LT GE GT IN
```

%QUPCASE is useful in the comparison of values because the macro facility does not automatically convert lowercase characters to uppercase before comparing values.

Comparisons

- %QUPCASE masks special characters and mnemonic operators while %UPCASE does not.

  %QUPCASE masks the same characters as the %NRBQUOTE function.

Example: Comparing %UPCASE and %QUPCASE

These statements show the results produced by %UPCASE and %QUPCASE:

```
%let a=begin;
%let b=%nrstr(&a);
%put UPCASE produces: %upcase(&b);
%put QUPCASE produces: %qupcase(&b);
```

When these statements execute, the following is written to the SAS log:

```
UPCASE produces: begin
QUPCASE produces: &A
```

See Also

"%UPCASE Macro Function" on page 371

%SCAN Macro Function

Searches for a word that is specified by its position in a string.

Type:	Macro function
See:	"%NRBQUOTE Macro Function" on page 320 and "%STR Macro Function" on page 342

Syntax

%SCAN(*character-string*, *n*<, *delimiters*<, *modifiers*>>)

Required Arguments

character-string
specifies a character string or a text expression that results in a character string. If the specified string might contain one or more special character and mnemonic operators, use %QSCAN. If the specified string contains a comma, enclose *character-string* in a quoting function such as %BQUOTE(*character-string*).

n
specifies an integer or a text expression that yields an integer, which specifies the position of the word to return. An implied %EVAL gives *n* numeric properties. If *n* is greater than the number of words in *argument*, the functions return a null string.

Note: When you are using Version 8 or greater, if *n* is negative, %SCAN examines the character string and selects the word that starts at the end of the string and searches backward.

Optional Arguments

delimiters

specifies an optional character expression that initializes a list of delimiters. This list determines which characters are used as the delimiters that separate words. The following rules apply:

- By default, all characters in *delimiters* are used as delimiters.
- If you specify the K modifier in the *modifier* argument, then all characters that are not in *delimiters* are used as delimiters.

Default	On ASCII systems: blank . < (+ & ! $ *) ; ^ - / , % |
Tip	You can add more characters to *delimiters* by using other modifiers.

modifiers

specifies a character constant, a variable, or an expression in which each non-blank character modifies the action of the %SCAN function. Blanks are ignored. You can use the following characters as modifiers:

a or A	adds alphabetic characters to the list of characters.
b or B	scans backward from right to left instead of from left to right, regardless of the sign of the *count* argument.
c or C	adds control characters to the list of characters.
d or D	adds digits to the list of characters.
f or F	adds an underscore and English letters (that is, valid first characters in a SAS variable name using VALIDVARNAME=V7) to the list of characters.
g or G	adds graphic characters to the list of characters. Graphic characters are characters that, when printed, produce an image on paper.
h or H	adds a horizontal tab to the list of characters.
i or I	ignores the case of the characters.
k or K	causes all characters that are not in the list of characters to be treated as delimiters. That is, if K is specified, then characters that are in the list of characters are kept in the returned value rather than being omitted because they are delimiters. If K is not specified, then all characters that are in the list of characters are treated as delimiters.
l or L	adds lowercase letters to the list of characters.
m or M	specifies that multiple consecutive delimiters, and delimiters at the beginning or end of the *string* argument, refer to words that have a length of zero. If the M modifier is not specified, then

	multiple consecutive delimiters are treated as one delimiter, and delimiters at the beginning or end of the *string* argument are ignored.
n or N	adds digits, an underscore, and English letters (that is, the characters that can appear in a SAS variable name using VALIDVARNAME=V7) to the list of characters.
o or O	processes the *charlist* and *modifier* arguments only once, rather than every time the %SCAN function is called. Using the O modifier in the DATA step (excluding WHERE clauses), or in the SQL procedure can make %SCAN run faster when you call it in a loop where the *charlist* and *modifier* arguments do not change. The O modifier applies separately to each instance of the %SCAN function in your SAS code, and does not cause all instances of the %SCAN function to use the same delimiters and modifiers.
p or P	adds punctuation marks to the list of characters.
q or Q	ignores delimiters that are inside substrings that are enclosed in quotation marks. If the value of the *string* argument contains unmatched quotation marks, then scanning from left to right will produce different words than scanning from right to left.
r or R	removes leading and trailing blanks from the word that %SCAN returns. If you specify both the Q and R modifiers, then the %SCAN function first removes leading and trailing blanks from the word. Then, if the word begins with a quotation mark, %SCAN also removes one layer of quotation marks from the word.
s or S	adds space characters to the list of characters (blank, horizontal tab, vertical tab, carriage return, line feed, and form feed).
t or T	trims trailing blanks from the *string* and *charlist* arguments. If you want to remove trailing blanks from only one character argument instead of both character arguments, then use the TRIM function instead of the %SCAN function with the T modifier.
u or U	adds uppercase letters to the list of characters.
w or W	adds printable (writable) characters to the list of characters.
x or X	adds hexadecimal characters to the list of characters.

Tip If the *modifier* argument is a character constant, then enclose it in quotation marks. Specify multiple modifiers in a single set of quotation marks. A *modifier* argument can also be expressed as a character variable or expression.

Details

The %SCAN and %QSCAN functions search *argument* and return the *n*th word. A word is one or more characters separated by one or more delimiters.

%SCAN does not mask special characters or mnemonic operators in its result, even when the argument was previously masked by a macro quoting function. %QSCAN masks the following special characters and mnemonic operators in its result:

```
& % ' " ( ) + − * / < > = ¬ ^ ~ ; , # blank
AND OR NOT EQ NE LE LT GE GT IN
```

Definition of “Delimiter” and “Word”

A delimiter is any of several characters that are used to separate words. You can specify the delimiters in the *charlist* and *modifier* arguments.

If you specify the Q modifier, then delimiters inside substrings that are enclosed in quotation marks are ignored.

In the %SCAN function, “word” refers to a substring that has all of the following characteristics:

- is bounded on the left by a delimiter or the beginning of the string
- is bounded on the right by a delimiter or the end of the string
- contains no delimiters

A word can have a length of zero if there are delimiters at the beginning or end of the string, or if the string contains two or more consecutive delimiters. However, the %SCAN function ignores words that have a length of zero unless you specify the M modifier.

Using Default Delimiters in ASCII Environments

If you use the %SCAN function with only two arguments, then the default delimiters are as follows on ASCII systems: `blank ! $ % & ( ) * + , - . / ; < ^¦`

If you use the *modifier* argument without specifying any characters as delimiters, then the only delimiters that will be used are delimiters that are defined by the *modifier* argument. In this case, the lists of default delimiters are not used. In other words, modifiers add to the list of delimiters that are specified by the *delimiters* argument. Modifiers do not add to the list of default modifiers.

Using the %SCAN Function with the M Modifier

If you specify the M modifier, then the number of words in a string is defined as one plus the number of delimiters in the string. However, if you specify the Q modifier, delimiters that are inside quotation marks are ignored.

If you specify the M modifier, then the %SCAN function returns a word with a length of zero if one of the following conditions is true:

- The string begins with a delimiter and you request the first word.
- The string ends with a delimiter and you request the last word.
- The string contains two consecutive delimiters and you request the word that is between the two delimiters.

Using the %SCAN Function without the M Modifier

If you do not specify the M modifier, then the number of words in a string is defined as the number of maximal substrings of consecutive non-delimiters. However, if you specify the Q modifier, delimiters that are inside quotation marks are ignored.

If you do not specify the M modifier, then the %SCAN function does the following:

- ignores delimiters at the beginning or end of the string
- treats two or more consecutive delimiters as if they were a single delimiter

If the string contains no characters other than delimiters, or if you specify a count that is greater in absolute value than the number of words in the string, then the %SCAN function returns one of the following:

- a single blank when you call the %SCAN function from a DATA step
- a string with a length of zero when you call the %SCAN function from the macro processor

Using Null Arguments

The %SCAN function allows character arguments to be null. Null arguments are treated as character strings with a length of zero. Numeric arguments cannot be null.

Comparisons

%QSCAN masks the same characters as the %NRBQUOTE function.

Example: Comparing the Actions of %SCAN and %QSCAN

This example illustrates the actions of %SCAN and %QSCAN.

```
%macro a;
   aaaaaa
%mend a;
%macro b;
   bbbbbb
%mend b;
%macro c;
   cccccc
%mend c;
%let x=%nrstr(%a*%b*%c);
%put X: &x;
%put The third word in X, with SCAN: %scan(&x,3,*);
%put The third word in X, with QSCAN: %qscan(&x,3,*);
```

The %PUT statement writes these lines to the log:

```
X: %a*%b*%c
The third word in X, with SCAN: cccccc
The third word in X, with QSCAN: %c
```

%STR Macro Function

Mask special characters and mnemonic operators in constant text at macro compilation.

Type:	Macro quoting function
Note:	The maximum level of nesting for macro quoting functions is 10.
See:	"%NRQUOTE Macro Function" on page 321

Syntax

%STR(*character-string*)

Required Argument

character-string

specifies a character string or an expression that results in a character string that contains special characters and mnemonic operators. The following special characters and mnemonic operators in the specified string are masked at macro compilation time:

```
+ - * / < > = ¬ ^ ~ ; ,  # blank
AND OR NOT EQ NE LE LT GE GT IN
```

Details

The %STR function masks a character string during compilation of a macro or macro language statement. It masks the following special characters and mnemonic operators:

```
+ - * / < > = ¬ ^ ~ ; ,  # blank
AND OR NOT EQ NE LE LT GE GT IN
```

They also mask the following characters when they occur in pairs and when they are not matched and are marked by a preceding `%`:

```
' " ( )
```

%STR does not mask the following characters:

```
& %
```

Table 17.3 Using %STR and %NRSTR Arguments

Argument	Use
Percent sign before a quotation mark - for example, %' or %",	Percent sign with quotation mark EXAMPLE: %let percent=%str(Jim%'s office);

Percent sign before a parenthesis - for example, %(or %)	Two percent signs (%%):
	EXAMPLE: %let x=%str(20%%);

Character string with the comment symbols /* or -->	%STR with each character EXAMPLE: %str(/) %str(*) *comment-text* %str(*)%str(/)

%STR is most useful for character strings that contain:

- a semicolon that should be treated as text rather than as part of a macro program statement
- blanks that are significant
- a quotation mark or parenthesis without a match
- no characters that require masking such as special characters and mnemonic operators.

CAUTION
Do not use %STR to enclose other macro functions or macro invocations that have a list of parameter values. Because %STR masks parentheses without a match, the macro processor does not recognize the arguments of a function or the parameter values of a macro invocation.

For a description of quoting in SAS macro language, see "Macro Quoting" on page 96.

Comparisons

- Of all the macro quoting functions, only %NRSTR and %STR take effect during compilation. The other macro quoting functions take effect when a macro executes.
- %STR masks the same items as %QUOTE. However, %QUOTE works during macro execution.
- %STR does not mask characters & and % while %NRSTR masks these characters.
- If resolution of a macro expression produces items that need to be masked, use the %BQUOTE or %NRBQUOTE function instead of the %STR or %NRSTR function.

Examples

Example 1: Maintaining Leading Blanks

This example enables the value of the macro variable TIME to contain leading blanks.

```
%let time=%str(   now);
%put Text followed by the value of time:&time;
```

When this example is executed, these lines are written to the SAS log:

```
Text followed by the value of time:   now
```

Example 2: Protecting a Blank So That It Will Be Compiled as Text

This example specifies that %QSCAN use a blank as the delimiter between words.

```
%macro words(string);
   %local count word;
   %let count=1;
   %let word=%qscan(&string,&count,%str( ));
   %do %while(&word ne);
      %let count=%eval(&count+1);
      %let word=%qscan(&string,&count,%str( ));
   %end;
   %let count=%eval(&count-1);
   %put The string contains &count words.;
%mend words;
%words(This is a very long string)
```

When this program executes, these lines are written to the SAS log:

```
The string contains 6 words.
```

Example 3: Quoting a Value That Might Contain a Macro Reference

The macro REVRS reverses the characters produced by the macro TEST. %NRSTR in the %PUT statement protects `%test&test` so that it is compiled as text and not interpreted.

```
%macro revrs(string);
   %local nstring;
   %do i=%length(&string) %to 1 %by -1;
      %let nstring=&nstring%qsubstr(&string,&i,1);
   %end;
 &nstring
%mend revrs;
%macro test;
   Two words
%mend test;
%put %nrstr(%test%test) - %revrs(%test%test);
```

When this program executes, the following lines are written to the SAS log:

```
1          %macro revrs(string);
2             %local nstring;
3             %do i=%length(&string) %to 1 %by -1;
4                %let nstring=&nstring%qsubstr(&string,&i,1);
5             %end;&nstring
6          %mend revrs;
7
8          %macro test;
9             Two words
10         %mend test;
11
12         %put %nrstr(%test%test) - %revrs(%test%test);
%test%test - sdrow owTsdrow owT
NOTE: SAS Institute Inc., SAS Campus Drive, Cary, NC USA 27513-2414
NOTE: The SAS System used:
      real time           0.28 seconds
      cpu time            0.12 seconds
```

%SUBSTR Macro Function

Produce a substring of a character string.

Type: Macro function

See: “%NRBQUOTE Macro Function” on page 320

Syntax

%SUBSTR(*character-string*, *position*<, *length*>)

Required Arguments

character-string
specifies a character string or an expression that results in a character string as the source string.

position
specifies an integer or an expression (text, logical, or arithmetic) that yields an integer, which specifies the position of the first character in the substring. If *position* is greater than the number of characters in the string, %SUBSTR issues a warning message and returns a null value. An automatic call to %EVAL causes *n* to be treated as a numeric value.

Optional Argument

length
specifies an optional integer or an expression (text, logical, or arithmetic) that yields an integer that specifies the number of characters in the substring. If *length* is greater than the number of characters following *position* in *character-*

string, %SUBSTR issues a warning message and returns a substring containing the characters from *position* to the end of the string. By default, %SUBSTR produces a string containing the characters from *position* to the end of the character string.

Details

The %SUBSTR function produces a substring of *character-string*, beginning at *position*, for *length* number of characters.

%SUBSTR does not mask special characters or mnemonic operators in its result, even when the argument was previously masked by a macro quoting function. %QSUBSTR masks the following special characters and mnemonic operators:

```
& % ' " ( ) + − * / < > = ¬ ^ ~ ; , #  blank
AND OR NOT EQ NE LE LT GE GT IN
```

Comparisons

%SUBSTR does not mask special characters and mnemonic operators. %QSUBSTR masks these characters.

Examples

Example 1: Limiting a Fileref to Eight Characters

The macro MAKEFREF uses %SUBSTR to assign the first eight characters of a parameter as a fileref, in case a user assigns one that is longer.

```
%macro makefref(fileref,file);
   %if %length(&fileref) gt 8 %then
       %let fileref = %substr(&fileref,1,8);
   filename &fileref "&file";
%mend makefref;
%makefref(humanresource,/dept/humanresource/report96)
```

SAS sees the following statement:

```
FILENAME HUMANRES "/dept/humanresource/report96";
```

Example 2: Comparing Actions of %SUBSTR and %QSUBSTR

Because the value of C is masked by %NRSTR, the value is not resolved at compilation. %SUBSTR produces a resolved result because it does not mask special characters and mnemonic operators in C before processing it, even though the value of C had previously been masked with the %NRSTR function.

```
%let a=one;
%let b=two;
```

```
%let c=%nrstr(&a &b);
%put C: &c;
%put With SUBSTR: %substr(&c,1,2);
%put With QSUBSTR: %qsubstr(&c,1,2);
```

When these statements execute, these lines are written to the SAS log:

```
C: &a &b
With SUBSTR: one
With QSUBSTR: &a
```

%SUPERQ Macro Function

Masks all special characters and mnemonic operators at macro execution but prevents further resolution of the value.

Type:	Macro quoting function
Note:	The maximum level of nesting for the macro quoting functions is 10.
See:	“%NRBQUOTE Macro Function” on page 320 and “%BQUOTE Macro Function” on page 314

Syntax

%SUPERQ(*macro-variable-name*)

Required Argument

macro-variable-name

specifies the name of a macro variable or an expression that results in the name of a macro variable without the leading ampersand. The following special characters and mnemonic operators in the value of the specified macro variable are masked:

```
& % ' " ( ) + – * / < > = ¬ ^ ~ ; , #  blank
AND OR NOT EQ NE LE LT GE GT IN
```

The resolution of other items in the value such as macro names and macro variable names is not performed.

Details

The %SUPERQ function returns the value of a macro variable without attempting to resolve any macros or macro variable references in the value. %SUPERQ masks the following special characters and mnemonic operators:

```
& % ' " ( ) + – * / < > = ¬ ^ ~ ; , #  blank
```

```
AND OR NOT EQ NE LE LT GE GT IN
```

%SUPERQ is particularly useful for masking macro variables that might contain an ampersand or a percent sign when they are used with the SYMPUT routine.

For a description of quoting in SAS macro language, see "Macro Quoting" on page 96.

Comparisons

- %SUPERQ is the only quoting function that prevents the resolution of macro variables and macro references in the value of the specified macro variable.
- %SUPERQ accepts only the name of a macro variable as its argument, without an ampersand, and the other quoting functions accept any text expression, including constant text, as an argument.
- %SUPERQ masks the same characters as the %NRBQUOTE function. However, %SUPERQ does not attempt to resolve anything in the value of a macro variable. %NRBQUOTE attempts to resolve any macro references or macro variable values in the argument before masking the result.

Example: Passing Unresolved Macro Variable Values

In this example, %SUPERQ prevents the macro processor from attempting to resolve macro references in the values of MV1 and MV2 before assigning them to macro variables TESTMV1 and TESTMV2.

```
data _null_;
   call symput('mv1','Smith&Jones');
   call symput('mv2','%macro abc;');
run;
%let testmv1=%superq(mv1);
%let testmv2=%superq(mv2);
%put Macro variable TESTMV1 is &testmv1;
%put Macro variable TESTMV2 is &testmv2;
```

When this program executes, these lines are written to the SAS log:

```
Macro variable TESTMV1 is Smith&Jones
Macro variable TESTMV2 is %macro abc;
```

You might think of the values of TESTMV1 and TESTMV2 as "pictures" of the original values of MV1 and MV2. The %PUT statement then writes the pictures in its text. The macro processor does not attempt resolution. It does not issue a warning message for the unresolved reference `&JONES` or an error message for beginning a macro definition inside a %LET statement.

%SYMEXIST Macro Function

Returns an indication of the existence of a macro variable.

Type:	Macro function

Syntax

%SYMEXIST(*macro-variable-name*)

Required Argument

macro-variable-name
specifies the name of a macro variable or an expression that results in the name of a macro variable without the leading ampersand.

Details

The %SYMEXIST function searches any enclosing local symbol tables and then the global symbol table for the indicated macro variable and returns one of the following values:

- `1` if the macro variable is found
- `0` if the macro variable is not found

Example: Using %SYMEXIST Macro Function

The following example uses the %IF %THEN %ELSE macro statement to change the value of `1` and `0` to **`TRUE`** and **`FALSE`** respectively:

```
%global x;
%macro test;
    %local y;
        %if %symexist(x) %then %put %nrstr(%symexist(x)) = TRUE;
                         %else %put %nrstr(%symexist(x)) = FALSE;
        %if %symexist(y) %then %put %nrstr(%symexist(y)) = TRUE;
                         %else %put %nrstr(%symexist(y)) = FALSE;
        %if %symexist(z) %then %put %nrstr(%symexist(z)) = TRUE;
                         %else %put %nrstr(%symexist(z)) = FALSE;
%mend test;
%test;
```

In the previous example, executing the %TEST macro writes the following output to the SAS log:

```
%symexist(x) = TRUE
%symexist(y) = TRUE
%symexist(z) = FALSE
```

%SYMGLOBL Macro Function

Returns an indication as to whether a macro variable is global in scope.

Type: Macro function

Syntax

%SYMGLOBL(*macro-variable-name*)

Required Argument

macro-variable-name
specifies the name of a macro variable or an expression that results in the name of a macro variable without the leading ampersand.

Details

The %SYMGLOBL function searches enclosing scopes for the indicated macro variable and returns a value of `1` if the macro variable is found in the global symbol table, otherwise it returns a `0`. For more information about the global and local symbol tables and macro variable scopes, see "Scopes of Macro Variables" on page 57.

Example: Using %SYMGLOBL Macro Function

The following example uses the %IF %THEN %ELSE macro statement to change the values of `1` and `0` to **`TRUE`** and **`FALSE`** respectively:

```
%global x;
    %macro test;
        %local y;
        %if %symglobl(x) %then %put %nrstr(%symglobl(x)) = TRUE;
                         %else %put %nrstr(%symglobl(x)) = FALSE;
        %if %symglobl(y) %then %put %nrstr(%symglobl(y)) = TRUE;
                         %else %put %nrstr(%symglobl(y)) = FALSE;
        %if %symglobl(z) %then %put %nrstr(%symglobl(z)) = TRUE;
                         %else %put %nrstr(%symglobl(z)) = FALSE;
%mend test;
```

```
%test;
```

In the example above, executing the %TEST macro writes the following output to the SAS log:

```
%symglobl(x) = TRUE
%symglobl(y) = FALSE
%symglobl(z) = FALSE
```

%SYMLOCAL Macro Function

Returns an indication as to whether a macro variable is local in scope.

Type: Macro function

Syntax

%SYMLOCAL(*macro-variable-name*)

Required Argument

macro-variable-name
specifies the name of a macro variable or a text expression that results in the name of a macro variable without the leading ampersand.

Details

The %SYMLOCAL searches enclosing scopes for the indicated macro variable and returns a value of 1 if the macro variable is found in a local symbol table, otherwise it returns a 0. For more information about the global and local symbol tables and macro variable scopes, see "Scopes of Macro Variables" on page 57.

Example: Using %SYMLOCAL Macro Function

The following example uses the %IF %THEN %ELSE macro statement to change the values of 1 and 0 to **TRUE** and **FALSE** respectively:

```
%global x;
%macro test;
    %local y;
        %if %symlocal(x) %then %put %nrstr(%symlocal(x)) = TRUE;
                         %else %put %nrstr(%symlocal(x)) = FALSE;
        %if %symlocal(y) %then %put %nrstr(%symlocal(y)) = TRUE;
                         %else %put %nrstr(%symlocal(y)) = FALSE;
        %if %symlocal(z) %then %put %nrstr(%symlocal(z)) = TRUE;
                         %else %put %nrstr(%symlocal(z)) = FALSE;
%mend test;
%test;
```

In the example above, executing the %TEST macro writes the following output to the SAS log:

```
%symlocal(x) = FALSE
%symlocal(y) = TRUE
%symlocal(z) = FALSE
```

%SYSEVALF Macro Function

Evaluates arithmetic and logical expressions using floating-point arithmetic.

Type:	Macro function
See:	"%EVAL Macro Function" on page 316

Syntax

%SYSEVALF(*expression*<, *conversion-type*>)

Required Argument

expression
specifies an arithmetic or logical expression to evaluate.

Restriction: The %SYSEVALF function does not support the IN logical operator in *expression*. However, you can use the %EVAL function in *expression* to evaluate an IN logical operator. See "Example 2: Using %EVAL to Evaluate an IN Logical Operator in a %SYSEVALF Expression" on page 355.

Optional Argument

conversion-type
converts the value returned by %SYSEVALF to the type of value specified. The value can then be used in other expressions that require a value of that type. *Conversion-type* can be one of the following:

BOOLEAN
returns

- 0 if the result of the expression is 0 or missing
- 1 if the result is any other value.

Here is an example:

```
%sysevalf(1/3,boolean)      /* returns 1 */
%sysevalf(10+.,boolean)     /* returns 0 */
```

CEIL

returns a character value representing the smallest integer that is greater than or equal to the result of the expression. If the result is within 10^{-12} of an integer, the function returns a character value representing that integer. An expression containing a missing value returns a missing value along with a message noting that fact:

```
%sysevalf(1 + 1.1,ceil)       /* returns  3 */
%sysevalf(-1 -2.4,ceil)       /* returns -3 */
%sysevalf(-1 + 1.e-11,ceil)   /* returns  0 */
%sysevalf(10+.)               /* returns  . */
```

FLOOR

returns a character value representing the largest integer that is less than or equal to the result of the expression. If the result is within 10^{-12} of an integer, the function returns that integer. An expression with a missing value produces a missing value:

```
%sysevalf(-2.4,floor)         /* returns -3 */
%sysevalf(3,floor)            /* returns  3 */
%sysevalf(1.-1.e-13,floor)    /* returns  1 */
%sysevalf(.,floor)            /* returns  . */
```

INTEGER

returns a character value representing the integer portion of the result (truncates the decimal portion). If the result of the expression is within 10^{-12} of an integer, the function produces a character value representing that integer. If the result of the expression is positive, INTEGER returns the same result as FLOOR. If the result of the expression is negative, INTEGER returns the same result as CEIL. An expression with a missing value produces a missing value:

```
%put %sysevalf(2.1,integer);        /* returns  2 */
%put %sysevalf(-2.4,integer);       /* returns -2 */
%put %sysevalf(3,integer);          /* returns  3 */
%put %sysevalf(-1.6,integer);       /* returns -1 */
%put %sysevalf(1.-1.e-13,integer);  /* returns  1 */
```

Details

About the %SYSEVALF Function

The %SYSEVALF function performs floating-point arithmetic and returns a value that is formatted using the BEST32. format. The result of the evaluation is always text. %SYSEVALF is the only macro function that can evaluate logical expressions that contain floating-point or missing values. Specify a conversion type to prevent problems when %SYSEVALF returns one of the following:

- missing or floating-point values to macro expressions

- macro variables that are used in other macro expressions that require an integer value

If the argument to the %SYSEVALF function contains no operator and no conversion type is specified, then the argument is returned unchanged.

For more information about evaluation of expressions by the SAS macro language, see Chapter 6, “Macro Expressions,” on page 85.

No Expression to Evaluate

If *expression* evaluates to a null value or one or more blank spaces, then there is nothing for %SYSEVALF to evaluate. In that case, the following error results:

```
ERROR: %SYSEVALF function has no expression to evaluate.
```

If you are using %SYSEVALF in your SAS code, verify that the expression in the offending %SYSEVALF macro call is correct. If you are not using %SYSEVALF in your code, this error might be the result of a SAS function call in your code. In some cases, this error can occur when calling SAS functions that have numeric and character arguments that can be specified in varying order. Examples of such functions include SUBSTRN, CATX, and FINDW. Refer to the function documentation for information about its arguments. For information about how to specify arguments for these functions, see “Specifying Argument Values That Can Be Character or Numeric” on page 358.

Comparisons

- %SYSEVALF supports floating-point numbers. However, %EVAL performs only integer arithmetic.
- You must use the %SYSEVALF macro function in macros to evaluate floating-point expressions. However, %EVAL is used automatically by the macro processor to evaluate macro expressions.

Examples

Example 1: Illustrating Floating-Point Evaluation

The macro FIGUREIT performs all types of conversions for SYSEVALF values.

```
%macro figureit(a,b);
   %let y=%sysevalf(&a+&b);
   %put The result with SYSEVALF is: &y;
   %put  The BOOLEAN value is: %sysevalf(&a +&b, boolean);
   %put  The CEIL value is: %sysevalf(&a +&b, ceil);
   %put  The FLOOR value is: %sysevalf(&a +&b, floor);
   %put  The INTEGER value is: %sysevalf(&a +&b, int);
%mend figureit;
%figureit(100,1.597)
```

When this program executes, these lines are written to the SAS log:

```
The result with SYSEVALF is: 101.597
The BOOLEAN value is: 1
The CEIL value is: 102
The FLOOR value is: 101
The INTEGER value is: 101
```

Example 2: Using %EVAL to Evaluate an IN Logical Operator in a %SYSEVALF Expression

Macro function %SYSEVALF does not support the IN logical operator in *expression*. However, you can use the %EVAL function to evaluate an IN logical operator in *expression*. This example demonstrates how this is done. In this example, macro %SHOWSALEPRICE accepts as arguments a product, the regular price of the product, and the discount that is applied to products that are on sale. Items that are currently on sale are laptops, desktops, and printers. By default, a 15% discount is applied to all on-sale products. If the specified product is on sale, the macro shows the discounted price and the amount of the discount. If the item is not on sale, the macro indicates that the product is not on sale and that the regular price applies.

Here is the SAS code.

```
%macro showSalePrice(product=, price=, discount=0.15) / minoperator;
   %let priceReduction =
      %sysevalf(%eval(%upcase(&product) in (LAPTOP DESKTOP PRINTER)) *
      &price * &discount);
   %if &priceReduction eq 0 %then %do;
      %put NOTE: %sysfunc(propcase(&product))s are not on sale today.;
      %put NOTE- Your price is %sysfunc(putn(&price, dollar9.2.)).;
   %end;
   %else %do;
      %let salePrice = %sysevalf(&price - &priceReduction);
      %put NOTE: Your sale price is %sysfunc(putn(&salePrice,
dollar9.2.)).;
      %put NOTE- You save %sysfunc(putn(&priceReduction, dollar6.))!;
   %end;
%mend showSalePrice;
```

The MINOPERATOR option in the %MACRO statement on page 406 specifies that the macro processor recognizes and evaluates the IN logical operator. In the first %SYSEVALF macro call, the %EVAL function is used to determine whether the specified product is on sale. The specified product is compared to the list of products that are on sale by using the IN logical operator. If the specified product is in the list, %EVAL returns 1, and priceReduction is calculated. If it is not in the list, %EVAL returns 0, and priceReduction is set to 0. The second %SYSEVALF macro call computes the product price. Here is an example for a laptop:

```
%showSalePrice(product=laptop,price=1899.99);
```

The following note is written to the SAS log:

```
NOTE: Your sale price is $1,614.99.
      You save   $285!
```

Here is an example for a paper shredder, which is not on sale:

```
%showSalePrice(product=shredder,price=149.99);
```

The following note is written to the SAS log:

```
NOTE: Shredders are not on sale today.
      Your price is   $149.99.
```

%SYSFUNC Macro Function

Execute SAS functions or user-written functions.

Type:	Macro function
Tip:	%SYSFUNC and %QSYSFUNC support SAS function names up to 32 characters.

Syntax

%SYSFUNC(*function(arguments)*<, *format*>)

Required Argument

function(arguments)

specifies the name of a function to execute and its arguments.

function

is the name of the function to execute. This function can be a SAS function, a function written with SAS/TOOLKIT software, or a function created using the "FCMP Procedure" in *Base SAS Procedures Guide*. The function cannot be a macro function.

All SAS functions, except those listed in Table 17.51 on page 359, can be used with %SYSFUNC.

You cannot nest functions to be used with a single %SYSFUNC. However, you can nest %SYSFUNC calls:

```
%let x=%sysfunc(trim(%sysfunc(left(&num))));
```

"Functions and Arguments for %SYSFUNC and %QSYSFUNC" on page 531 shows the syntax of SAS functions used with %SYSFUNC.

arguments

specifies one or more arguments used by *function*.

An argument can be a macro variable reference or a text expression that produces arguments for a function.

See "Specifying Function Arguments" on page 357

Optional Argument

format

specifies an optional format to apply to the result of *function*. This format can be provided by SAS, generated by PROC FORMAT, or created with SAS/TOOLKIT. There is no default value for *format*. If you do not specify a *format*, the SAS macro facility does not perform a *format* operation on the result and uses the default of the *function*.

Details

Specifying Function Arguments

Specifying Character Argument Values

Because %SYSFUNC is a macro function, you do not need to enclose character values in quotation marks as you do in DATA step functions. For example, the arguments to the OPEN function are enclosed in quotation marks when the function is used alone in a DATA step statement, but do not require quotation marks when used within %SYSFUNC. These statements show the difference:

- function OPEN used in DATA step statements:

  ```
  dsid=open("Sasuser.Houses","i");
  dsid=open("&mydata","&mode");
  ```

- function OPEN used within %SYSFUNC:

  ```
  %let dsid = %sysfunc(open(Sasuser.Houses,i));
  %let dsid=%sysfunc(open(&mydata,&mode));
  ```

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

Note: The arguments to %SYSFUNC are evaluated according to the rules of the SAS macro language. This includes both the function name and the argument list to the function. In particular, an empty argument position will not generate a NULL argument, but a zero-length argument.

%SYSFUNC does not mask special characters and mnemonic operators, which include:

```
& % ' " ( ) + - * / < > = ¬ ^ ~ ; , #  blank
AND OR NOT EQ NE LE LT GE GT IN
```

If *arguments* might contain any of these characters, use %QSYSFUNC instead. See "%QSYSFUNC Macro Function" on page 329.

Specifying Numeric Argument Values

When a function called by %SYSFUNC requires a numeric argument, %SYSFUNC converts the argument to a numeric value. The value can be a number, an expression that evaluates to a number, or a function that returns a number. Here is an example.

```
%put Result: %sysfunc(mean(83, 6 * 2, %sysfunc(divide(50,2))));
```

```
Result: 40
```

Specifying Argument Values That Can Be Character or Numeric

Some SAS functions such as SUBSTRN and CATX accept a numeric or character value as one or more arguments. For these arguments, %SYSFUNC first silently attempts to convert the argument value to a numeric value using the %SYSEVALF on page 352 function. If the conversion is successful, the numeric value is then passed to the called function as the argument. If the %SYSEVALF function fails, %SYSFUNC passes the argument value as-is to the called function. Here is a simple example.

```
%put %sysfunc(catx(%str( ), Result:, (1 + (5 * 8)) / 4));
```

The first argument in the CATX function is a constant string, so %SYSFUNC passes the value (blank) to the CATX function. The subsequent CATX function arguments can be numeric or character. In this example, %SYSFUNC first attempts to silently convert the second argument value to a numeric value using the %SYSEVALF function. Because `%SYSEVALF(Result:)` fails, %SYSFUNC passes the value `Result:` to the CATX function as the second argument value. When %SYSFUNC attempts to convert the third argument to a numeric value, the %SYSEVALF function returns 10.25. %SYSFUNC then passes 10.25 to the CATX function as the third argument value. Here is the result:

```
Result: 10.25
```

In some cases, this behavior can cause unexpected results. Consider the following example.

```
%put %sysfunc(catx(%str( ), Wind:, NE, 15-25 MPH));
```

```
Wind: 0 15-25 MPH
```

In this case, when %SYSFUNC evaluates the second argument value, %SYSEVALF(NE) returns 0 because NE is the not-equals mnemonic operator, and null NE null is False. %SYSFUNC passes 0 to the CATX function as the second argument value instead of NE. Similar issues can happen when argument values contain mathematical symbols.

To prevent NE from being evaluated to 0 in this example, use nested %NRSTR on page 323 and %STR on page 342 functions as follows:

```
%put %sysfunc(catx(%str( ), Wind:, %nrstr(%str(NE)), 15-25 MPH));
```

The %NRSTR function masks the %STR function result in such a way that the %STR function result is evaluated inside the CATX function. This ensures that the literal NE is passed in as the second argument instead of 0. Here is the result:

```
Wind: NE 15-25 MPH
```

If an argument cannot be evaluated by %SYSEVALF, an error results, but execution continues and the argument is passed to the called function. This can occur if an argument is one or more blank spaces or is a null value. In either case, there is no expression for %SYSEVALF to evaluate. Here is a simple example:

```
%put %sysfunc(cat(A, %str( ), B));
```

Because the second argument is a single space, %SYSEVALF has nothing to evaluate, so it writes the following message to the SAS log:

```
ERROR: %SYSEVALF function has no expression to evaluate.
```

However, execution continues and the space is passed to the CAT function, which then correctly concatenates the strings:

```
A B
```

In this case, to eliminate the error, use the CATX function instead and specify a blank space as the delimiter as shown in the following example:

```
%put %sysfunc(catx(%str( ), A, B));
```

The result is the same.

SAS Functions That Are Not Supported

The following table lists the SAS functions that are not available with macro function %SYSFUNC.

Table 17.4 *SAS Functions Not Available with %SYSFUNC and %QSYSFUNC*

All Variable Information Functions	ALLCOMB	ALLPERM
DIF	DIM	HBOUND
INPUT	IORCMSG	LAG
LBOUND	LEXCOMB	LEXCOMBI
LEXPERK	LEXPERM	METADATA_GETNOBJ
MISSING	PUT	RESOLVE
SYMGET		
All SAS File I/O Functions	MD5	SHA256

Note: INPUT and PUT are not available with %SYSFUNC. Use INPUTN, INPUTC, PUTN, and PUTC instead.

Note: The Variable Information functions include functions such as VNAME and VLABEL. For a complete list, see “Definitions of Functions and CALL Routines” in *SAS Functions and CALL Routines: Reference*.

Function Result

%SYSFUNC does not mask the following special characters or mnemonic operators in its result:

```
& % ' " ( ) + − * / < > = ¬ ^ ~ ; , #  blank
AND OR NOT EQ NE LE LT GE GT IN
```

If the function result contains one or more of these characters, use %QSYSFUNC on page 329 instead. %SYSFUNC can return a floating point number when the function that it executes supports floating point numbers.

CAUTION
Values returned by SAS functions might be truncated. Although values returned by macro functions are not limited to the length imposed by the DATA step, values returned by SAS functions do have that limitation.

Comparisons

%SYSFUNC does not mask special characters and mnemonic operators. %QSYSFUNC masks the same characters and mnemonic operators as the %NRBQUOTE on page 320 function.

Examples

Example 1: Formatting the Current Date in a TITLE Statement

This example formats a TITLE statement containing the current date using the DATE function and the WORDDATE. format:

```
title "%sysfunc(date(),worddate.) Absence Report";
```

When the program is executed on July 18, 2008, the statement produces the following TITLE statement:

```
title "July 18, 2008 Absence Report"
```

Example 2: Formatting a Value Produced by %SYSFUNC

In this example, the TRY macro transforms the value of PARM using the PUTN function and the CATEGORY. format.

```
   proc format;
     value category
     Low-<0   = 'Less Than Zero'
     0        = 'Equal To Zero'
     0<-high  = 'Greater Than Zero'
     other    = 'Missing';
   run;
   %macro try(parm);
     %put &parm is %sysfunc(putn(&parm,category.));
   %mend;
   %try(1.02)
   %try(.)
   %try(-.38)
```

When these statements are executed, these lines are written to the SAS log:

```
1.02 is Greater Than Zero
. is Missing
-.38 is Less Than Zero
```

Example 3: Translating Characters

%SYSFUNC executes the TRANSLATE function to translate the Ns in a string to Ps.

```
%let string1 = V01N01-V01N10;
%let string1 = %sysfunc(translate(&string1,P, N));
%put With N translated to P, V01N01-V01N10 is &string1;
```

When these statements are executed, these lines are written to the SAS log:

```
With N translated to P, V01N01-V01N10 is V01P01-V01P10
```

Example 4: Confirming the Existence of a SAS Data Set

The macro CHECKDS uses %SYSFUNC to execute the EXIST function, which checks the existence of a data set:

```
%macro checkds(dsn);
   %if %sysfunc(exist(&dsn)) %then
      %do;
         proc print data=&dsn;
         run;
      %end;
      %else
         %put The data set &dsn does not exist.;
%mend checkds;
%checkds(Sasuser.Houses)
```

See Example Code 10.1 on page 151. When the program is executed, the following statements are produced:

```
PROC PRINT DATA=SASUSER.HOUSES;
RUN;
```

Example 5: Determining the Number of Variables and Observations in a Data Set

Many solutions have been generated in the past to obtain the number of variables and observations present in a SAS data set. Most past solutions have used a combination of _NULL_ DATA steps, SET statement with NOBS=, and arrays to obtain this information. Now, you can use the OPEN and ATTRN functions to obtain this information quickly and without interfering with step boundary conditions.

```
%macro obsnvars(ds);
   %global dset nvars nobs;
   %let dset=&ds;
   %let dsid = %sysfunc(open(&dset));
   %if &dsid %then
      %do;
         %let nobs =%sysfunc(attrn(&dsid,NOBS));
         %let nvars=%sysfunc(attrn(&dsid,NVARS));
         %let rc = %sysfunc(close(&dsid));
```

```
            %put &dset has &nvars  variable(s) and &nobs observation(s).;
         %end;
      %else
         %put Open for data set &dset failed - %sysfunc(sysmsg());
   %mend obsnvars;
   %obsnvars(Sasuser.Houses)
```

See Example Code 10.1 on page 151. When the program is executed, the following message appears in the SAS log:

```
sasuser.houses has 6 variable(s) and 15 observation(s).
```

%SYSGET Macro Function

Returns the character string that is the value of the environment variable passed as the argument.

Type:	Macro function
Note:	Both Linux and SAS environment variables can be translated using the %SYSGET function.
See:	

Syntax

%SYSGET(*environment-variable*)

Required Argument

environment-variable
specifies the name of an environment variable. The case of *environment-variable* must agree with the case that is stored on the operating environment. if *environment-variable* does not exist, an error message is written to the SAS log.

Details

The %SYSGET function returns the value as a character string. If the value is truncated or the variable is not defined on the operating environment, %SYSGET displays a warning message in the SAS log.

You can use the value returned by %SYSGET as a condition for determining further action to take or parts of a SAS program to execute. For example, your program can restrict certain processing or issue commands that are specific to a user.

Examples

Example 1: Using SYSGET to Get the ID of the Current User

This example returns the value of the USER environment variable:

```
%let person=%sysget(USER);
%put User is &person;
```

When these statements execute for user ABCDEF, the following is written to the SAS log:

```
User is abcdef
```

Example 2: Using SYSGET to Get the Home Directory

This example returns the value of the HOME environment variable:

```
%let home=%sysget(HOME);
%put Home path is &home;
```

Here is an example of what is written to the SAS log:

```
Home path is /users/sasdemo
```

%SYSMACEXEC Macro Function

Returns an indication of the execution status of a macro.

Type:	Macro function

Syntax

%SYSMACEXEC(*macro-name*)

Required Argument

macro-name
specifies the name of a macro or an expression that results in the name of a macro without the leading percent sign.

Details

The %SYSMACEXEC function returns 1 if the specified macro is currently executing. Otherwise, it returns 0.

%SYSMACEXIST Macro Function

Returns an indication of the existence of a macro definition in the Work.SASMacr or Work.SASMac*n* catalog.

Type:	Macro function
Note:	The Work.Sasmacr catalog is used to store compiled macros for the primary SAS session. In applications or programs that use side sessions, the catalog used to store compiled macros for each side session is Work.Sasmac*n*, where *n* is a unique integer.

Syntax

%SYSMACEXIST(*macro-name*)

Required Argument

macro-name
specifies the name of a macro or an expression that results in the name of a macro without the leading percent sign.

Details

The %SYSMACEXIST function returns 1 if a definition for the specified macro exists in the Work.SASMacr or Work.SASMac*n* catalog. Otherwise, it returns 0.

%SYSMEXECDEPTH Macro Function

Returns the nesting depth of macro execution from the point of the call to %SYSMEXECDEPTH.

Type:	Macro Function
Tip:	%SYSMEXECDEPTH and %SYSMEXECNAME were implemented to be used together, but it is not required.
See:	%SYSMEXECNAME Function

Syntax

%SYSMEXECDEPTH

Details

To retrieve the nesting level of the currently executing macro, use the %SYSMEXECDEPTH. This function returns a number indicating the depth of the macro in nested macro calls. The following are the %SYSMEXECDEPTH return value descriptions:

0	open code
>0	nesting level

See the following example and explanation that follow it.

Examples

Example 1

The following program shows how to use %SYSMEXECDEPTH and %SYSMEXECNAME to track the depth of macro execution. See the explanation that follows the program.

```
%macro A;
   %put %nrstr(%%)sysmexecdepth=%sysmexecdepth;
%mend A;

/* The macro execution depth of a macro called from open
   code is one.  */
%A;

%macro B;
   %put %nrstr(%%)sysmexecdepth=%sysmexecdepth;
   %put %nrstr(%%)sysmexecname(1)=%sysmexecname(1);
   %put %nrstr(%%)sysmexecname(2)=%sysmexecname(2);
   %put %nrstr(%%)sysmexecname(0)=%sysmexecname(0);
   %put %nrstr(%%)sysmexecname(%nrstr(%%)sysmexecdepth-1)=
   %sysmexecname(%sysmexecdepth-1);
%mend B;

%macro C;
   %B;
%mend;

%C;
```

The following is written to the SAS log:

```
%sysmexecdepth=2
%sysmexecname(1)=C
%sysmexecname(2)=B
%sysmexecname(0)=OPEN CODE
%sysmexecname(%sysmexecdepth-1)=C
```

Here is a description of the execution:

- Macro **A** calls macro **B**. Macro **C** calls macro **B**. A call to %SYSMEXECDEPTH placed in macro **C** would return the value **2** for macro **B**.
- If the macro **C** wanted to know the name of the macro that had called it, it could call %SYSMEXECNAME with `%SYSMEXECNAME(%SYSMEXECDEPTH-1` (the value of the *n* argument being %SYSMEXECDEPTH, its own nesting level, minus one). That call to %SYSMEXECNAME would return the value B.

Example 2

The following program shows another way to use %SYSMEXECDEPTH and %SYSMEXECNAME to track the depth of macro execution.

```
%macro level1;
   %level2;
%mend;

%macro level2;
   %level3;
%mend;

%macro level3;
   %level4;
%mend;

%macro level4;
   %do i = %sysmexecdepth %to 0 %by -1;
      %put %nrstr(%%)sysmexecname(&i)=%sysmexecname(&i);
   %end;
%mend;

%level1;
```

The following is written to the SAS log:

```
%sysmexecname(4)=LEVEL4
%sysmexecname(3)=LEVEL3
%sysmexecname(2)=LEVEL2
%sysmexecname(1)=LEVEL1
%sysmexecname(0)=OPEN CODE
```

%SYSMEXECNAME Macro Function

Returns the name of the macro executing at a requested nesting level.

Type:	Macro Function
Tip:	%SYSMEXECNAME and %SYSMEXECDEPTH were implemented to be used together, but it is not required.
See:	%SYSMEXECDEPTH function

Syntax

%SYSMEXECNAME(*nesting-level*)

Required Argument

nesting-level
specifies the nesting level at which you are requesting the macro name.

0
specifies open code

>0
specifies a nesting level

Details

The %SYSMEXECNAME function returns the name of the macro executing at the *n* nesting level. The following three scenarios are shown in the example below.

- If *n = 0*, **`open code`** is returned.
- If *n >%SYSMEXECDEPTH*, a null string is returned and a WARNING diagnostic message is issued to the SAS log.
- If *n<0*, a null string is returned and a WARNING diagnostic message is issued to the SAS log.

Example

```
%put %sysmexecdepth;
%put %sysmexecname(%sysmexecdepth);
%put %sysmexecname(%sysmexecdepth + 1);
%put %sysmexecname(%sysmexecdepth - 1);
```

The following is written to the SAS log:

```
1    %put %sysmexecdepth;
0
2    %put %sysmexecname(%sysmexecdepth);
OPEN CODE
3    %put %sysmexecname(%sysmexecdepth + 1);
WARNING: Argument 1 to %SYSMEXECNAME function is out of range.

4    %put %sysmexecname(%sysmexecdepth - 1);
WARNING: Argument 1 to %SYSMEXECNAME function is out of range.
```

%SYSPROD Macro Function

Reports whether a SAS software product is licensed at the site.

Type:	Macro function
See:	“%SYSEXEC Macro Statement” on page 424, “SYSSCP and SYSSCPL Automatic Macro Variable” on page 270, and “SYSVER Automatic Macro Variable” on page 277

Syntax

%SYSPROD(*product-code*)

Required Argument

product-code

specifies a SAS software product code. *Product-code* can be a character string or text expression that yields a code for a SAS product.

Table 17.5 Commonly Used Product Codes

AF	CPE	GRAPH	PH-CLINICAL
ASSIST	EIS	IML	QC
BASE	ETS	INSIGHT	SHARE
CALC	FSP	LAB	STAT
CONNECT	GIS	OR	TOOLKIT

For codes for other SAS software products, contact your on-site SAS support personnel.

Details

%SYSPROD can return the values shown in the following table.

Table 17.6 %SYSPROD Values and Descriptions

Value	Description
1	The SAS product is licensed.
0	The SAS product is not licensed.
-1	The product is not Institute software (for example, if the product code is misspelled).

Example: Verifying the SAS/GRAPH License Before Running the GPLOT Procedure

This example uses %SYSPROD to determine whether to execute a PROC GPLOT statement or a PROC SGPLOT statement, based on whether SAS/GRAPH software has been licensed.

```
%macro runplot(ds);
   %if %sysprod(graph)=1 %then
      %do;
         title "GPLOT of %upcase(&ds)";
         proc gplot data=&ds;
            plot style*price / haxis=0 to 150000 by 50000;
         run;
         quit;
      %end;
   %else
      %do;
         title "SGPLOT of %upcase(&ds)";
         proc sgplot data=&ds;
            scatter x = price y = style;
         run;
         quit;
      %end;
%mend runplot;

%runplot(Sasuser.Houses)
```

See Example Code 10.1 on page 151. When this program executes and SAS/GRAPH is licensed, the following statements are generated:

```
TITLE "GPLOT of SASUSER.HOUSES";
PROC GPLOT DATA=SASUSER.HOUSES;
PLOT STYLE*PRICE / HAXIS=0 TO 150000 BY 50000;
RUN;
```

%UNQUOTE Macro Function

During macro execution, unmasks all special characters and mnemonic operators for a value.

Type:	Macro function
See:	"%BQUOTE Macro Function" on page 314, "%NRBQUOTE Macro Function" on page 320, "%NRQUOTE Macro Function" on page 321, "%NRSTR Macro Function" on page 323, "%QUOTE Macro Function" on page 334, "%STR Macro Function" on page 342, and "%SUPERQ Macro Function" on page 347

Syntax

%UNQUOTE(*character-string*)

Required Argument

character-string
specifies a character string or an expression that results in a character string.

Details

The %UNQUOTE function unmasks a value so that special characters that it might contain are interpreted as macro language elements instead of as text. The most important effect of %UNQUOTE is to restore normal tokenization of a value whose tokenization was altered by a previous macro quoting function. %UNQUOTE takes effect during macro execution.

For more information, see "Macro Quoting" on page 96.

Example: Using %UNQUOTE to Unmask Values

This example demonstrates a problem that can arise when the value of a macro variable is assigned using a macro quoting function and then the variable is referenced in a later DATA step. If the value is not unmasked before it reaches the SAS compiler, the DATA step does not compile correctly and it produces error messages. Although several macro functions automatically unmask values, a variable might not be processed by one of those functions.

The following program generates error messages in the SAS log because the value of TESTVAL is still masked when it reaches the SAS compiler.

```
%let val = aaa;
%let testval = %str(%'&val%');
```

```
data _null_;
  val = &testval;
  put 'VAL =' val;
run;
```

This version of the program runs correctly because %UNQUOTE unmasks the value of TESTVAL.

```
%let val = aaa;
%let testval = %str(%'&val%');
data _null_;
  val = %unquote(&testval);
  put 'VAL =' val;
run;
```

This program prints the following to the SAS log:

```
VAL=aaa
```

%UPCASE Macro Function

Convert values to uppercase.

Type:	Macro function
Tip:	To convert characters to lowercase, use the %LOWCASE or %QLOWCASE autocall macro.
See:	“%LOWCASE Autocall Macro” on page 219, “%NRBQUOTE Macro Function” on page 320, and “%QLOWCASE Autocall Macro” on page 222

Syntax

%UPCASE(*character-string*)

Required Argument

character-string
specifies a character string or an expression that results in a character string that is to be converted to upper case.

Details

The %UPCASE function converts lowercase characters in the argument to uppercase. %UPCASE does not mask special characters or mnemonic operators in its result, even when the argument was previously masked by a macro quoting function. If the specified string contains one or more special characters and mnemonic operators, use %QUPCASE instead. See "%QUPCASE Macro Function" on page 336.

%UPCASE is useful in the comparison of values because the macro facility does not automatically convert lowercase characters to uppercase before comparing values.

Comparisons

- %UPCASE does not mask special characters and mnemonic operators while %QUPCASE does.

Examples

Example 1: Capitalizing a Value to Be Compared

In this example, the macro REPTNOTE compares a value input for the macro variable MONTH to the string DECEMBER. If the uppercase value of the response is DECEMBER, then note "Year-end report" is output is written to the SAS log. Otherwise, note "Report for *month*" is written to the log.

```
%macro reptNote(month);
   %if %upcase(&month) = DECEMBER %then
       %put NOTE: Year-end report;
   %else %put NOTE: Report for &month;
%mend reptNote;
```

You can invoke the macro in any of these ways to satisfy the %IF condition:

```
%reptNote(DECEMBER)
%reptNote(December)
%reptNote(december)
```

Example 2: Comparing %UPCASE and %QUPCASE

These statements show the results produced by %UPCASE and %QUPCASE:

```
%let a=begin;
%let b=%nrstr(&a);
%put UPCASE produces: %upcase(&b);
%put QUPCASE produces: %qupcase(&b);
```

When these statements execute, the following is written to the SAS log:

```
UPCASE produces: begin
QUPCASE produces: &A
```

18

SQL Clauses for Macros

SQL Clauses for Macros

Structured Query Language (SQL) is a standardized, widely used language for retrieving and updating data in databases and relational tables.

Dictionary

INTO Clause

Assigns values produced by PROC SQL to macro variables.

Type:	SELECT statement, PROC SQL

Syntax

INTO : *macro-variable-specification-1* <, : *macro-variable-specification-2* ...>

Required Argument

macro-variable-specification

names one or more macro variables to create or update. Precede each macro variable name with a colon (:). The macro variable specification can be in any one or more of the following forms:

: *macro-variable*

specify one or more macro variables. Leading and trailing blanks are not trimmed from values before they are stored in macro variables:

```
select style, sqfeet
   into :type, :size
   from sasuser.houses;
```

:*macro-variable-1* – : *macro-variable-n* <NOTRIM>
:*macro-variable-1* THROUGH : *macro-variable-n* <NOTRIM>
:*macro-variable-1* THRU : *macro-variable-n* <NOTRIM>

specifies a numbered list of macro variables. Leading and trailing blanks are trimmed from values before they are stored in macro variables. If you do not want the blanks to be trimmed, use the NOTRIM option. NOTRIM is an option in each individual element in this form of the INTO clause, so you can use it on one element and not on another element:

```
select style, sqfeet
   into :type1 - :type4 notrim, :size1 - :size3
   from sasuser.houses;
```

:*macro-variable* SEPARATED BY '*characters* ' <NOTRIM>

specifies one macro variable to contain all the values of a column. Values in the list are separated by one or more *characters*. This form of the INTO clause is useful for building a list of items. Leading and trailing blanks are trimmed from values before they are stored in the macro variable. If you do not want the blanks to be trimmed, use the NOTRIM option. You can use the DISTINCT option in the SELECT statement to store only the unique column (variable) values:

```
select distinct style
   into :types separated by ','
   from sasuser.houses;
```

Details

The INTO clause for the SELECT statement can assign the result of a calculation or the value of a data column (variable) to a macro variable. If the macro variable does not exist, INTO creates it. You can check the PROC SQL macro variable SQLOBS to see the number of rows (observations) produced by a SELECT statement.

The INTO clause can be used only in the outer query of a SELECT statement and not in a subquery. The INTO clause cannot be used when you are creating a table (CREATE TABLE) or a view (CREATE VIEW).

Macro variables created with INTO follow the scoping rules for the %LET statement. For more information, see “%LET Macro Statement” on page 402.

Values assigned by the INTO clause use the BEST8. format.

Comparisons

In the SQL procedure, the INTO clause performs a role similar to the SYMPUT routine.

Examples

The following examples use data set Sasuser.Houses. See Example Code 10.1 on page 151.

Example 1: Storing Column Values in Declared Macro Variables

This example is based on the data set SASUser.Houses and stores the values of columns (variables) STYLE and SQFEET from the first row of the table (or observation in the data set) in macro variables TYPE and SIZE. The %LET statements strip trailing blanks from TYPE and leading blanks from SIZE because this type of specification with INTO does not strip those blanks by default.

```
proc sql noprint;
   select style, sqfeet
      into :type, :size
      from sasuser.houses;
quit;
%let type=&type;
%let size=&size;
%put The first row contains a &type with &size square feet.;
```

When this program executes, the following is written to the SAS log:

```
The first row contains a RANCH with 1250 square feet.
```

Example 2: Storing Row Values in a List of Macro Variables

This example creates two lists of macro variables, TYPE1 through TYPE4 and SIZE1 through SIZE4, and stores values from the first four rows (observations) of the SASUser.Houses data set in them. The NOTRIM option for TYPE1 through TYPE4 retains the trailing blanks for those values.

```
proc sql noprint;
   select style, sqfeet
      into :type1 - :type4 notrim, :size1 - :size4
      from sasuser.houses;
quit;
%macro putit;
   %do i=1 %to 4;
      %put Row&i: Type=**&&type&i**   Size=**&&size&i**;
   %end;
%mend putit;
```

```
%putit
```

When this program executes, these lines are written to the SAS log:

```
Row1: Type=**RANCH   **  Size=**1250**
Row2: Type=**SPLIT   **  Size=**1190**
Row3: Type=**CONDO   **  Size=**1400**
Row4: Type=**TWOSTORY**  Size=**1810**
```

Example 3: Storing Values of All Rows in One Macro Variable

This example stores all values of the column (variable) STYLE in the macro variable TYPES and separates the values with a comma and a blank.

```
proc sql;
   select distinct quote(style)
      into :types separated by ', '
      from sasuser.houses;
quit;
%put Types of houses=&types.;
```

When this program executes, this line is written to the SAS log:

```
Types of houses="CONDO ", "RANCH ", "SPLIT ", "TWOSTORY"
```

19

Macro Statements

Macro Statements

A macro language statement instructs the macro processor to perform an operation. It consists of a string of keywords, SAS names, and special characters and operators, and it ends in a semicolon. Some macro language statements are used only in macro definitions, but you can use others anywhere in a SAS session or job, either inside or outside macro definitions (referred to as open code).

Dictionary

%ABORT Macro Statement

Stops the macro that is executing along with the current DATA step, SAS job, or Compute Server.

Type:	Macro statement
Restriction:	Allowed in macro definitions only

Syntax

%ABORT <ABEND | CANCEL | EXIT | RETURN | <*n*> | <FILE>>;

Required Arguments

ABEND

causes abnormal termination of the current macro and SAS job. For the Compute Server, the SAS compute session is terminated. An abnormal error code is reflected in the `sessionConditionCode`. The session state is marked as `canceled`. Results depend on the method of operation:

- batch mode and non-interactive mode.
  - stops processing immediately.
  - sends an error message to the SAS log that states that execution was terminated by the ABEND option of the %ABORT macro statement.
  - does not execute any subsequent statements or check syntax.
  - returns control to the operating environment. Further action is based on how your operating environment and your site treat jobs that end abnormally.

- interactive line mode.
  - causes your macro interactive line mode to stop processing immediately and return you to your operating environment.

CANCEL <FILE>

causes the cancellation of the current submitted statements. The results depend on the method of operation.

If the method of operation is batch mode and non-interactive mode, use the CANCEL option to do the following:

- The entire SAS program and SAS are terminated.
- The error message is written to the SAS log.

If the method of operation is interactive line mode, use the CANCEL option to do the following:

- It clears only the currently submitted program.
- Other subsequent submitted programs are not affected.
- The error message is written to the SAS log.

If the method of operation is on the Compute Server, use the CANCEL option to do the following:

- It clears only the currently submitted program.
- Other subsequent submit calls are not affected.
- The error message is written to the SAS log.

If the method of operation is SAS/IntrNet application server, use the CANCEL option to do the following:

- A separate execution is created for each request. The execution submits the request code. A CANCEL in the request code clears the current submitted code but does not terminate the execution or the SAS Compute Server.

FILE

when coded as an option to the CANCEL argument in an autoexec file or in a %INCLUDE file, causes only the contents of the autoexec file or %INCLUDE file to be cleared by the %ABORT statement. Other submitted source statements will be executed after the autoexec or %INCLUDE file.

Restriction: The CANCEL argument cannot be submitted using SAS/SHARE, SAS/CONNECT, or SAS/AF.

CAUTION: **When the %ABORT CANCEL FILE option is executed within a %INCLUDE file, all open macros are closed and execution resumes at the next source line of code.**

RETURN

causes abnormal termination of the current macro and SAS job or session. For the Compute Server, the SAS compute session is terminated. An abnormal error code is reflected in the `sessionConditionCode`. The session state is marked as `canceled`. Results depend on the method of operation:

- batch mode and non-interactive mode

  - stops processing immediately
  - sends an error message to the SAS log that states that execution was terminated by the RETURN option of the %ABORT macro statement
  - does not execute any subsequent statements or check syntax
  - returns control to the operating environment with a condition code indicating an error
- interactive line mode
  - causes your macro and interactive line mode to stop processing immediately and return you to your operating environment

n

an integer value that enables you to specify a condition code:

- When used with the CANCEL argument, the value is placed in the SYSINFO automatic macro variable.
- When it is NOT used with the CANCEL statement, SAS returns the value to the operating environment when the execution stops. The range of values for *n* depends on your operating environment.

Details

If you specify no argument, the %ABORT macro statement produces these results under the following methods of operation:

- batch mode and non-interactive mode.
  - stops processing the current macro and DATA step and writes an error message to the SAS log. Data sets can contain an incomplete number of observations or no observations, depending on when SAS encountered the %ABORT macro statement.
  - sets the OBS= system option to 0.
  - continues limited processing of the remainder of the SAS job, including executing macro statements, executing system option statements, and syntax checking of program statements.
- interactive line mode
  - stops processing the current macro and DATA step. Any further DATA steps or procedures execute normally.

Comparisons

The %ABORT macro statement causes SAS to stop processing the current macro and DATA step. What happens next depends on

- the method that you use to submit your SAS statements
- the arguments that you use with %ABORT

- your operating environment

The %ABORT macro statement usually appears in a clause of an %IF-%THEN macro statement that is designed to stop processing when an error condition occurs.

Note: The return code generated by the %ABORT macro statement is ignored by SAS if the system option ERRORABEND is in effect.

Note: When you execute an %ABORT macro statement in a DATA step, SAS does not use data sets that were created in the step to replace existing data sets with the same name.

Example

This example shows how to use the %ABORT statement in a macro to stop execution when an error is detected. The macro prints data in a data set. It accepts as arguments the name of the data set and optionally, the number of observations to print. By default, 5 observations are printed.

```
%macro printData(name, obs=5);
   %if &name eq %then %do;
      %put ERROR: You must specify a data set name.;
      %abort cancel;
   %end;
   %else %if %sysfunc(exist(&name)) eq 0 %then %do;
      %put ERROR: Data set &name not found.;
      %abort cancel;
   %end;

   proc print data=&name(obs=&obs);
   run;
%mend printData;
```

If a data set name is not specified or if the data set specified does not exist, the macro prints an error message, and then executes the %ABORT statement to stop execution. In the following code, because the required data set name is not specified in the %printData macro call, the %ABORT statement is executed. When the %ABORT statement is executed, the macro execution is stopped, and the %PUT statement that follows the %printData macro call is canceled.

```
%printData;  /* This statement causes %printData to execute %ABORT. */
%put %nrstr(NOTE: This %PUT statement is canceled when %ABORT is
executed.);
```

```
ERROR: You must specify a data set name.
ERROR: Execution canceled by an %ABORT CANCEL statement.
NOTE: The SAS System stopped processing due to receiving a CANCEL request.
```

When valid input is specified, the data is printed and the %PUT statement that follows is executed.

```
%printData(sashelp.cars);
%put %nrstr(NOTE: This %PUT statement is canceled when %ABORT is
executed.);
```

```
NOTE: There were 5 observations read from the data set SASHELP.CARS.
NOTE: PROCEDURE PRINT used (Total process time):
      real time           0.08 seconds
      cpu time            0.03 seconds

NOTE: This %PUT statement is canceled when %ABORT is executed.
```

%* Macro Comment Macro Statement

Designates comment text.

Type:	Macro statement
Restriction:	Allowed in macro definitions or open code

Syntax

%*_commentary_;

Required Argument

commentary
is a descriptive message of any length.

Details

The macro comment statement is useful for describing macro code. Text from a macro comment statement (*commentary*) is not constant text and is not stored in a compiled macro. Because a semicolon ends the macro comment statement, the comment cannot contain internal semicolons unless the internal semicolons are enclosed in quotation marks. All single and double quotation marks within the comment must have a closing quotation mark. Unmatched quotation marks within the comment cause problems. Macro comment statements are not recognized when they are enclosed in quotation marks.

Macro comment statements are complete macro statements and are processed by the macro facility. Quotation marks within a macro comment must match.

Only macro comment statements and PL/1-style comments in the form `/* commentary */` in macro definitions or open code can be used to hide macro statements from processing by the macro facility.

Comparisons

In open code, the following SAS comment statements are complete SAS statements:

```
*commentary;
```

or

```
comment commentary;
```

Consequently, they are processed by the tokenizer and macro facility and cannot contain semicolons or unmatched quotation marks.

In a macro definition, the following SAS comment statements are stored as constant text in the compiled macro:

```
*commentary;
```

or

```
comment commentary;
```

If *commentary* contains macro statements, the following rules apply:

- The macro statements that are embedded in *commentary* are executed regardless of the asterisk or COMMENT, which can result in unexpected behavior.
- Two semicolons are required, one to end the macro statements and one to end the comment. If only one semicolon is used, the comment continues to the next statement in the program and ends with its semicolon, which can result in syntax errors or unexpected behavior.

Using asterisk-type or COMMENT-type comments within a macro definition is not recommended.

SAS comments in the following PL/1 form are not tokenized, but are processed as a string of characters:

```
/*commentary*/
```

These comments can appear anywhere that a single blank can appear. The comment can contain semicolons, special characters, macro triggers such as & and %, or unmatched quotation marks without causing problems. SAS comments in the PL/1 form are not stored in a compiled macro.

TIP PL/1-style comments are the safest comments to use.

For more information about using comments in macro definitions, see SAS KB0036213: Using comments within a macro on support.sas.com.

Example: Contrasting Comment Types

This example defines and invokes the macro VERDATA, which checks the data for values that exceed a specified threshold. It contains asterisk-style, PL/1-style, and macro-style comments to demonstrate the differences. This example reads data from input file ina.txt, which includes the following data:

```
0  1  2
3  4  5
6  7  8
9  10 11
12 13 14
```

Here is the SAS code for this example.

```
filename ina "ina.txt";
data _null_;
   file ina;
   put "0  1  2";
   put "3  4  5";
   put "6  7  8";
   put "9  10 11";
   put "12 13 14";
run;

%macro verdata(in, thresh);
   *%let thresh = 2;;                  /* 1 */

   /* %let thresh = 5; */              /* 2 */
   %if %length(&in) > 0 %then %do;
      %* infile given;                 /* 3 */
      %put NOTE: Threshold is &thresh..;
      data check;
         /* Jim's data */              /* 4 */
         infile &in;
         input x y z;
         * check data;                 /* 5 */
         if x<&thresh or y<&thresh or z<&thresh then list;
      run;
   %end;
   %else %put Error: No infile specified;
%mend verdata;

%verdata(ina, 9)
filename ina clear;
```

Here is an explanation of the comments that are used in this macro definition.

1 This asterisk-style comment includes macro statement %LET. Notice that two semicolons terminate the comment. The first semicolon terminates the %LET statement, and the second terminates the comment. If only one semicolon is used, the comment includes the DATA CHECK statement and ends with its semicolon, which causes syntax errors. Regardless of the asterisk, the %LET statement is executed, which causes unexpected results. This is demonstrated later.

2 This PL/1-style comment prevents embedded macro statement %LET from executing. This can also be accomplished with a macro-style comment:

```
%*%let thresh = 5;
```

3 This macro-style comment prevents the INFILE GIVEN statement from executing.

4 This PL/1-style comment contains informational text.

5 This asterisk-style comment contains informational text only, which can be used in a macro definition. However, a PL/1-style or macro-style comment is recommended.

When you execute %VERDATA, the macro processor generates the following:

```
MPRINT(VERDATA):   ;
MPRINT(VERDATA):   data check;
MPRINT(VERDATA):   infile ina;
MPRINT(VERDATA):   input x y z;
MPRINT(VERDATA):   * check data;
MPRINT(VERDATA):   if x<2 or y<2 or z<2 then list;
MPRINT(VERDATA):   run;
```

However, the result is not as expected:

```
NOTE: Threshold is 2.
…
RULE:      ----+----1----+----2 …
1          0  1  2 7
```

The threshold should have been 9, and the first 3 observations should have been listed. This result is because the %LET statement in the first asterisk-style comment in the macro definition was executed, which changed THRESH to 2. To get the correct result, remove that comment or change it to a PL/1-style comment or a macro-style comment. Here is the correct result.

```
NOTE: Threshold is 9.
…
RULE:      ----+----1----+----2 …
1          0  1  2 7
2          3  4  5 7
3          6  7  8 7
```

%COPY Macro Statement

Copies specified items from a SAS macro library.

Type:	Macro statement
Restriction:	Allowed in macro definitions or open code
See:	"%MACRO Macro Statement" on page 406 and "SASMSTORE= System Option" on page 475

Syntax

%COPY *macro-name* / <LIBRARY=*libref*> < OUTFILE=*fileref* | '*external file*'> SOURCE;

Required Argument

macro-name
name of the macro that the %COPY statement will use.

Optional Arguments

LIBRARY=*libref*
specifies the libref of a SAS library that contains a catalog of stored compiled SAS macros. If no library is specified, the libref specified by the SASMSTORE= option is used.

Restriction: This libref cannot be Work.

Alias LIB=

OUTFILE=*fileref* | '*external file*'
specifies the output destination of the %COPY statement. The value can be a fileref or an external file.

Alias OUT=

Required Argument That Follows "/"

SOURCE
specifies that the source code of the macro will be copied to the output destination. If the OUTFILE= option is not specified, the source is written to the SAS log.

Alias SRC

Requirement You must specify SOURCE.

Example: Using %COPY Statement

In the following example, the %COPY statement writes the stored source code to the SAS log:

```
/* commentary */ %macro foobar(arg) /store source
     des="This macro does not do much";
%put arg = &arg;
* this is commentary!!!;
%* this is macro commentary;
%mend /* commentary; */;      /* Further commentary */
```

```
%copy foobar/source;
```

The following results are written to the SAS log:

```
%macro foobar(arg) /store source
des="This macro does not do much";
%put arg = &arg;
* this is commentary!!!;
%* this is macro commentary;
%mend /* commentary; */;
```

%DO Macro Statement

Begins a %DO group.

Type:	Macro statement
Restriction:	Allowed in macro definitions. Allowed in open code when used only with %IF-%THEN/%ELSE statements.
See:	"%END Macro Statement" on page 393

Syntax

%DO;
text and macro language statements

%END;

Details

The %DO statement designates the beginning of a section of a macro definition that is treated as a unit until a matching %END statement is encountered. This macro section is called a %DO group. %DO groups can be nested.

A simple %DO statement often appears in conjunction with %IF-%THEN/%ELSE statements to designate a section of the macro to be processed depending on whether the %IF condition is true or false.

Example: Producing One of Two Reports

This macro uses two %DO groups with the %IF-%THEN/%ELSE statement to conditionally print one of two reports.

```
%macro reportit(request);
   %if %upcase(&request)=STAT %then
```

```
         %do;
            proc means;
               title "Summary of All Numeric Variables";
            run;
         %end;
      %else %if %upcase(&request)=PRINTIT %then
         %do;
            proc print;
               title "Listing of Data";
            run;
         %end;
      %else %put Incorrect report type. Please try again.;
      title;
   %mend reportit;
   %reportit(stat)
   %reportit(printit)
```

Specifying `stat` as a value for the macro variable REQUEST generates the PROC MEANS step. Specifying `printit` generates the PROC PRINT step. Specifying any other value writes a customized error message to the SAS log.

%DO macro-variable=start %TO stop Macro Statement

Executes a section of a macro repetitively based on the value of an index variable.

Type:	Macro statement
Restriction:	Allowed in macro definitions only
See:	"%END Macro Statement" on page 393

Syntax

%DO *macro-variable=start* %TO *stop* <%BY *increment*>;
text and macro language statements

%END;

Required Arguments

macro-variable

names a macro variable or a text expression that generates a macro variable name. Its value functions as an index that determines the number of times the %DO loop iterates. If the macro variable specified as the index does not exist, the macro processor creates it in the local symbol table.

You can change the value of the index variable during processing. For example, using conditional processing to set the value of the index variable beyond the *stop* value when a certain condition is met ends processing of the loop.

start
specifies an integer (or a macro expression that generates an integer) to use as the initial value of *macro-variable*. *Start* and *stop* control the number of times the statements between the iterative %DO statement and the %END statement are processed. For the first iteration, *macro-variable* is equal to *start*. As processing continues, the value of *macro-variable* changes by the value of *increment* until the value of *macro-variable* is outside the range of integers specified by *start* and *stop*.

%TO *stop*
specifies an integer (or a macro expression that generates an integer) to use as the final *macro-variable* iteration value. Iteration stops if the value of *macro-variable* exceeds this value.

Optional Argument

%BY *increment*
specifies a nonzero integer (or a macro expression that generates a nonzero integer) to be added to the value of the index variable in each iteration of the loop. By default, *increment* is 1. *Increment* is evaluated before the first iteration of the loop. Therefore, you cannot change it as the loop iterates.

Example: Generating a Series of DATA Steps

This example illustrates using an iterative %DO group in a macro definition.

```
%macro create(howmany);
   %do i=1 %to &howmany;
      data month&i;
         infile in&i;
         input product cost date;
      run;
   %end;
%mend create;
%create(3)
```

When you execute the macro CREATE, it generates these statements:

```
DATA MONTH1;
   INFILE IN1;
   INPUT PRODUCT COST DATE;
RUN;
DATA MONTH2;
   INFILE IN2;
   INPUT PRODUCT COST DATE;
RUN;
DATA MONTH3;
   INFILE IN3;
   INPUT PRODUCT COST DATE;
RUN;
```

%DO %UNTIL Macro Statement

Executes a section of a macro repetitively until a condition is true.

Type:	Macro statement
Restriction:	Allowed in macro definitions only
See:	"%END Macro Statement" on page 393

Syntax

%DO %UNTIL(*expression*);
text and macro language statements

%END;

Required Argument

expression
can be any macro expression that resolves to a logical value. The macro processor evaluates the expression at the bottom of each iteration. The expression is true if it is an integer other than zero. The expression is false if it has a value of zero. If the expression resolves to a null value or a value containing nonnumeric characters, the macro processor issues an error message.

These examples illustrate expressions for the %DO %UNTIL statement:

```
%do %until(&hold=no);

%do %until(%index(&source,&excerpt)=0);
```

Details

The %DO %UNTIL statement checks the value of the condition at the bottom of each iteration. Thus, a %DO %UNTIL loop always iterates at least once.

Example: Validating a Parameter

This example uses the %DO %UNTIL statement to scan an option list to test the validity of the parameter TYPE.

```
%macro grph(type);
   %let type=%upcase(&type);
   %let options=DOT HBAR VBAR;
   %let i=0;
```

```
      %do %until (&type=%scan(&options,&i) or (&i>3)) ;
         %let i = %eval(&i+1);
      %end;
      %if &i>3 %then %do;
         %put ERROR: &type type not supported;
      %end;
      %else %do;
         proc sgplot data=sashelp.cars;
         &type type / group=origin;
         run;
      %end;
   %mend grph;
```

When you invoke the GRPH macro with a value of HBAR, the macro generates these statements:

```
PROC CHART;
HBAR SEX / GROUP=DEPT;
RUN;
```

If you invoke the GRPH macro with a value of PIE, then the %PUT statement writes this line to the SAS log:

```
ERROR: PIE type not supported
```

%DO %WHILE Macro Statement

Executes a section of a macro repetitively while a condition is true.

Type:	Macro statement
Restriction:	Allowed in macro definitions only
See:	"%END Macro Statement" on page 393

Syntax

%DO %WHILE (*expression*);
text and macro program statements

%END;

Required Argument

expression
can be any macro expression that resolves to a logical value. The macro processor evaluates the expression at the top of each iteration. The expression is true if it is an integer other than zero. The expression is false if it has a value of zero. If the expression resolves to a null value or to a value containing nonnumeric characters, the macro processor issues an error message.

These examples illustrate expressions for the %DO %WHILE statement:

```
%do %while(&a<&b);

%do %while(%length(&name)>20);
```

Details

The %DO %WHILE statement tests the condition at the top of the loop. If the condition is false the first time the macro processor tests it, the %DO %WHILE loop does not iterate.

Example: Removing Markup Tags from a Title

This example demonstrates using the %DO %WHILE to strip markup (SGML) tags from text to create a TITLE statement:

```
%macro untag(title);
   %let stbk=%str(<);
   %let etbk=%str(>);
   /* Do loop while tags exist  */
   %do %while (%index(&title,&stbk)>0) ;
      %let pretag=;
      %let posttag=;
      %let pos_et=%index(&title,&etbk);
      %let len_ti=%length(&title);
      /* Is < first character? */
      %if (%qsubstr(&title,1,1)=&stbk) %then %do;
         %if (&pos_et ne &len_ti) %then
            %let posttag=%qsubstr(&title,&pos_et+1);
      %end;
      %else %do;
         %let pretag=%qsubstr(&title,1,(%index(&title,&stbk)-1));
         /* More characters beyond end of tag (>) ? */
         %if (&pos_et ne &len_ti) %then
            %let posttag=%qsubstr(&title,&pos_et+1);
      %end;
      /* Build title with text before and after tag */
      %let title=&pretag&posttag;
   %end;
   title "&title";
%mend untag;
```

You can invoke the macro UNTAG as

```
%untag(<title>Total <emph>Overdue </emph>Accounts</title>)
```

The macro then generates this TITLE statement:

```
TITLE "Total Overdue Accounts";
```

If the title text contained special characters such as commas, you could invoke it with the %NRSTR function.

```
%untag(
   %nrstr(<title>Accounts: Baltimore, Chicago, and Los Angeles</
title>))
```

%END Macro Statement

Ends a %DO group.

Type:	Macro statement
Restriction:	Allowed in macro definitions. Allowed in open code when used only with %DO in %IF-%THEN/%ELSE statements.

Syntax

%END;

Example: Ending a %DO Group

This macro definition contains a %DO %WHILE loop that ends, as required, with a %END statement:

```
%macro test(finish);
   %let i=1;
   %do %while (&i<&finish);
      %put the value of i is &i;
      %let i=%eval(&i+1);
   %end;
%mend test;
%test(5)
```

Invoking the TEST macro with 5 as the value of *finish* writes these lines to the SAS log:

```
The value of i is 1
The value of i is 2
The value of i is 3
The value of i is 4
```

%GLOBAL Macro Statement

Creates macro variables that are available during the execution of an entire Compute Server session.

Type:	Macro statement

Restriction:	Allowed in macro definitions or open code
See:	“%LOCAL Macro Statement” on page 403

Syntax

%GLOBAL *macro-variable(s)*;

Or

%GLOBAL / READONLY *macro-variable=value*;

Required Argument

macro-variable(s)
is the name of one or more macro variables or a text expression that generates one or more macro variable names. You cannot use a SAS variable list or a macro expression that generates a SAS variable list in a %GLOBAL statement.

Optional Argument

READONLY *macro-variable=value*
creates a new read only global macro variable.

Note: The READONLY option can be used to create a single new macro variable (local or global).

macro-variable
is the name of a macro variable or a text expression that produces a macro variable name. The name must be a new macro variable name.

value
is a character string or a text expression.

TIP Omitting value produces a null value (0 characters).

TIP Leading and trailing blanks in the value are ignored. To have leading and trailing blanks contained in the value, you must enclose the value within the %STR function.

Details

The %GLOBAL statement creates one or more global macro variables and assigns null values to the variables. Global macro variables are variables that are available during the entire execution of the Compute Server session or job.

A macro variable created with a %GLOBAL statement has a null value until you assign it some other value. If a global macro variable already exists and you specify that variable in a %GLOBAL statement, the existing value remains unchanged.

%GLOBAL statements that use the READONLY option create a new global macro variable and assign the specified value. Existing macro variables cannot be made read-only. The value of the global macro variable cannot be changed and the variable cannot be deleted. A macro variable that is declared with the READONLY option cannot be re-declared in the same scope or any enclosed scope. All read-only macro variables persist until the scope in which they exist is deleted.

Comparisons

- Both the %GLOBAL statement and the %LOCAL statement create macro variables with a specific scope. However, the %GLOBAL statement creates global macro variables that exist for the duration of the session or job. The %LOCAL statement creates local macro variables that exist only during the execution of the macro that defines the variable.
- If you define both a global macro variable and a local macro variable with the same name, the macro processor uses the value of the local variable during the execution of the macro that contains the local variable. When the macro that contains the local variable is not executing, the macro processor uses the value of the global variable.

Example: Creating Global Variables in a Macro Definition

```
%macro vars(first=1,last=);
   %global gfirst glast;
   %let gfirst=&first;
   %let glast=&last;
   var test&first-test&last;
%mend vars;
```

When you submit the following program, the macro VARS generates the VAR statement and the values for the macro variables used in the title statement.

```
proc print;
   %vars(last=50)
   title "Analysis of Tests &gfirst-&glast";
run;
```

SAS sees the following:

```
PROC PRINT;
   VAR TEST1-TEST50;
   TITLE "Analysis of Tests 1-50";
RUN;
```

%GOTO Macro Statement

Branches macro processing to the specified label.

Type:	Macro statement
Alias:	%GO TO
Restriction:	Allowed in macro definitions only
See:	"%label Macro Statement" on page 400

Syntax

%GOTO *label*;

Required Argument

label

is either the name of the label that you want execution to branch to or a text expression that generates the label. A text expression that generates a label in a %GOTO statement is called a *computed %GOTO destination*.[1]

The following examples illustrate how to use *label*:

```
%goto findit;  /* branch to the label FINDIT */

%goto &home;   /* branch to the label that is */
               /* the value of the macro variable HOME */
```

CAUTION

No percent sign (%) precedes the label name in the %GOTO statement. The syntax of the %GOTO statement does not include a % in front of the label name. If you use a %, the macro processor attempts to call a macro by that name to generate the label.

Details

Branching with the %GOTO statement has two restrictions. First, the label that is the target of the %GOTO statement must exist in the current macro; you cannot branch to a label in another macro with a %GOTO statement. Second, a %GOTO statement cannot cause execution to branch to a point inside an iterative %DO, %DO %UNTIL, or %DO %WHILE loop that is not currently executing.

1. A computed %GOTO contains % or & and resolves to a label.

Example: Providing Exits in a Large Macro

The %GOTO statement is useful in large macros when you want to provide an exit if an error occurs.

```
%macro check(parm);
   %local status;
   %if &parm= %then %do;
       %put ERROR:  You must supply a parameter to macro CHECK.;
       %goto exit;
   %end;
   more macro statements that test for error conditions
   %if &status > 0 %then %do;
       %put ERROR:  File is empty.;
       %goto exit;
   %end;
   more macro statements that generate text
   %put Check completed successfully.;
%exit: %mend check;
```

%IF-%THEN/%ELSE Macro Statement

Conditionally process a portion of a macro.

Type:	Macro statement
Restrictions:	Allowed in macro definitions or in open code.
	No text, other than a comment, is allowed between the semicolon that ends the ACTION and the %ELSE statement.
	In open code, the ACTION that is associated with both the %THEN and %ELSE statements must be a %DO statement.
	In open code, %IF-%THEN/%ELSE statements cannot be nested.

Syntax

%IF *expression* **%THEN** *action*;< %ELSE *action*;>

Required Arguments

%IF expression

is any macro expression that resolves to an integer. If the expression resolves to an integer other than zero, the expression is true and the %THEN clause is processed. If the expression resolves to zero, then the expression is false and the %ELSE statement, if one is present, is processed. If the expression resolves to a null value or a value containing nonnumeric characters, the macro processor issues an error message. For more information about writing macro expressions and their evaluation, see Chapter 6, “Macro Expressions,” on page 85 .

The following examples illustrate using expressions in the %IF-%THEN statement:

-
  ```
  %if &name=GEORGE %then %let lastname=smith;
  ```

-
  ```
  %if %upcase(&name)=GEORGE %then %let lastname=smith;
  ```

-
  ```
  %if &i=10 and &j>5 %then %put check the index variables;
  ```

%THEN *action*

is either constant text, a text expression, or a macro statement. If *action* contains semicolons (for example, in SAS statements), then the first semicolon after %THEN ends the %THEN clause. Use a %DO group or a quoting function, such as %STR, to prevent semicolons in *action* from ending the %IF-%THEN statement. The following examples show two ways to conditionally generate text that contains semicolons:

-
  ```
  %if &city ne %then %do;
                 keep citypop statepop;
              %end;
           %else %do;
                 keep statepop;
              %end;
  ```

-
  ```
  %if &city ne %then %str(keep citypop statepop;);
              %else %str(keep statepop;);
  ```

Details

The macro language does not contain a subsetting %IF statement. Thus, you cannot use %IF without %THEN.

Expressions that compare character values in the %IF-%THEN statement use the sort sequence of the host operating system for the comparison. For more information about host sort sequences, see "SORT Procedure" in *Base SAS Procedures Guide*.

No text, other than a comment, is allowed between the semicolon that ends the ACTION and the %ELSE statement. When the following example executes, the extra semicolon is treated as text. Therefore, an error message is written to the SAS log:

```
%if &city ne %then %do;
              keep citypop statepop;
           %end; ;
        %else %do;
              keep statepop;
           %end;
```

Comparisons

Although they look similar, the %IF-%THEN/%ELSE statement and the IF-THEN/ELSE statement belong to two different languages. In general, %IF-%THEN/%ELSE statement, which is part of the SAS macro language, conditionally generates text. However, the IF-THEN/ELSE statement, which is part of the SAS language, conditionally executes SAS statements during DATA step execution.

The expression that is the condition for the %IF-%THEN/%ELSE statement can contain only operands that are constant text or text expressions that generate text. However, the expression that is the condition for the IF-THEN/ELSE statement can contain only operands that are DATA step variables, character constants, numeric constants, or date and time constants.

When the %IF-%THEN/%ELSE statement generates text that is part of a DATA step, it is compiled by the DATA step compiler and executed. On the other hand, when the IF-THEN/ELSE statement executes in a DATA step, any text generated by the macro facility has been resolved, tokenized, and compiled. No macro language elements exist in the compiled code. “Example 1: Contrasting the %IF-%THEN/%ELSE Statement with the IF-THEN/ELSE Statement” illustrates this difference.

For more information, see “SAS Programs and Macro Processing” on page 17 and Chapter 6, “Macro Expressions,” on page 85.

Examples

Example 1: Contrasting the %IF-%THEN/%ELSE Statement with the IF-THEN/ELSE Statement

In the SETTAX macro, the %IF-%THEN/%ELSE statement tests the value of the macro variable TAXRATE to control the generation of one of two DATA steps. The first DATA step contains an IF-THEN/ELSE statement that uses the value of the DATA step variable SALE to set the value of the DATA step variable TAX.

```
%macro settax(taxrate);
   %let taxrate = %upcase(&taxrate);
   %if &taxrate = CHANGE %then
      %do;
         data thisyear;
            set lastyear;
            if  sale > 100 then tax = .05;
            else tax = .08;
         run;
      %end;
   %else %if &taxrate = SAME %then
      %do;
         data thisyear;
            set lastyear;
            tax = .03;
            run;
      %end;
%mend settax;
```

If the value of the macro variable TAXRATE is `CHANGE`, then the macro generates the following DATA step:

```
DATA THISYEAR;
   SET LASTYEAR;
   IF SALE > 100 THEN TAX = .05;
   ELSE TAX = .08;
RUN;
```

If the value of the macro variable TAXRATE is `SAME`, then the macro generates the following DATA step:

```
DATA THISYEAR;
   SET LASTYEAR;
   TAX = .03;
RUN;
```

Example 2: Conditionally Printing Reports

In this example, the %IF-%THEN/%ELSE statement generates statements to produce one of two reports.

```
%macro fiscal(report);
   %if %upcase(&report)=QUARTER %then
      %do;
         title 'Quarterly Revenue Report';
         proc means data=total;
            var revenue;
         run;
      %end;
   %else
      %do;
         title 'To-Date Revenue Report';
         proc means data=current;
            var revenue;
         run;
      %end;
%mend fiscal;
%fiscal(quarter)
```

When invoked, the macro FISCAL generates these statements:

```
TITLE 'Quarterly Revenue Report';
PROC MEANS DATA=TOTAL;
VAR REVENUE;
RUN;
```

%label Macro Statement

Identifies the destination of a %GOTO statement.

Type:	Macro statement
Restriction:	Allowed in macro definitions only
See:	"%GOTO Macro Statement" on page 396

Syntax

%label: *macro-text*

Required Arguments

label
specifies a SAS name.

macro-text
is a macro statement, a text expression, or constant text. The following examples illustrate each:

- `%one: %let book=elementary;`
- `%out: %mend;`
- `%final: data _null_;`

Details

- The label name is preceded by a %. When you specify this label in a %GOTO statement, do not precede it with a %.
- An alternative to using the %GOTO statement and statement label is to use a %IF-%THEN statement with a %DO group.

Example: Controlling Program Flow

In the macro INFO, the %GOTO statement causes execution to jump to the label QUICK when the macro is invoked with the value of `short` for the parameter TYPE.

```
data economy;
   set sashelp.cars;
   avgMpg = (MPG_highway + MPG_City)/2;
   if (avgMpg >= 32);
run;

%macro info(type);
   %if %upcase(&type)=SHORT %then %goto quick; /* No % here */
      proc contents;
      run;
      proc freq;
         tables _numeric_;
      run;
   %quick: proc print data=_last_(obs=5);      /* Use % here */
      run;
%mend info;

%info(short)
```

Invoking the macro INFO with TYPE equal to `short` generates these statements:

```
PROC PRINT DATA=_LAST_(OBS=5);
   RUN;
```

%LET Macro Statement

Creates a macro variable and assigns it a value.

Type:	Macro statement
Restriction:	Allowed in macro definitions or open code
See:	“%STR Macro Function” on page 342

Syntax

%LET *macro-variable*=<*value*>;

Required Arguments

macro-variable
is either the name of a macro variable or a text expression that produces a macro variable name. The name can refer to a new or existing macro variable.

value
is a character string or a text expression. Omitting *value* produces a null value (0 characters). Leading and trailing blanks in *value* are ignored. To make them significant, enclose *value* with the %STR function.

Details

If the macro variable that is named in the %LET statement already exists in any enclosing scope, the %LET statement updates the value. If the macro variable that is named in the %LET statement does not exist, it is created in the closest enclosing scope and it is assigned the specified value. A %LET statement can define only one macro variable at a time.

Example: Sample %LET Statements

This example illustrates several %LET statements:

```
%macro title(text,number);
    %put TITLE&number "&text";
%mend;
%let topic=  The History of Genetics  ; /* Leading and trailing */
```

```
                                            /* blanks are removed   */
%title(&topic,1)
%let subject=topic;                         /* &subject resolves    */
%let &subject=Genetics Today;               /* before assignment    */
%title(&topic,2)
%let subject=The Future of Genetics;        /* &subject resolves    */
%let topic= &subject;                       /* before assignment    */
%title(&topic,3)
```

When you submit these statements, the TITLE statements that are generated by the TITLE macro are written to the SAS log:

```
13   %macro title(text,number);
14       %put TITLE&number "&text";
15   %mend;
16   %let topic=  The History of Genetics   ;
17
18   %title(&topic,1)
TITLE1 "The History of Genetics"
19   %let subject=topic;
20   %let &subject=Genetics Today;
21   %title(&topic,2)
TITLE2 "Genetics Today"
22   %let subject=The Future of Genetics;
23   %let topic= &subject;
24   %title(&topic,3)
TITLE3 "The Future of Genetics"
```

%LOCAL Macro Statement

Creates macro variables that are available only during the execution of the macro where they are defined.

Type:	Macro statement
Restriction:	Allowed in macro definitions only
See:	“%GLOBAL Macro Statement” on page 393

Syntax

%LOCAL *macro-variable(s)*;

Or

%LOCAL / READONLY *macro-variable*=*value*;

Required Argument

macro-variable(s)

is the name of one or more macro variables or a text expression that generates one or more macro variable names. You cannot use a SAS variable list or a macro expression that generates a SAS variable list in a %LOCAL statement.

Optional Argument

READONLY *macro-variable=value*

creates a new read-only local macro variable.

Note: The READONLY option can be used to create a single new macro variable (local or global).

macro-variable

is the name of a macro variable or a text expression that produces a macro variable name.

Requirement	The macro variable name must be unique, regardless of scope. If you declare a local macro variable with the READONLY option and a macro variable of the same name already exists in global scope or in a nested local scope, an error results.

value

is a character string or a text expression.

TIP Omitting value produces a null value (0 characters).

TIP Leading and trailing blanks in the value are ignored. To include leading and trailing blanks in the value, you must enclose the value within the %STR function.

Details

The %LOCAL statement creates one or more local macro variables. A macro variable created with %LOCAL has a null value until you assign it some other value. Local macro variables are variables that are available only during the execution of the macro in which they are defined.

Use the %LOCAL statement to ensure that macro variables created earlier in a program are not inadvertently changed by values assigned to variables with the same name in the current macro. If a local macro variable already exists and you specify that variable in a %LOCAL statement, the existing value remains unchanged.

%LOCAL statements that use the READONLY option create a new local macro variable and assign the specified value. Existing macro variables cannot be made read-only. The value of the local macro variable cannot be changed and the variable cannot be deleted. A macro variable that is declared with the READONLY option cannot be re-declared in the same scope or any enclosed scope. When declaring a new local macro variable with the READONLY option, the name of the new macro variable must be unique, regardless of scope. Otherwise, an error results. All read-only macro variables persist until the scope in which they exist is deleted.

Comparisons

- Both the %LOCAL statement and the %GLOBAL statement create macro variables with a specific scope. However, the %LOCAL statement creates local macro variables that exist only during the execution of the macro that contains the variable, and the %GLOBAL statement creates global macro variables that exist for the duration of the session or job.
- If you define a local macro variable and a global macro variable with the same name, the macro facility uses the value of the local variable during the execution of the macro that contains that local variable. When the macro that contains the local variable is not executing, the macro facility uses the value of the global variable.

Example: Using a Local Variable with the Same Name as a Global Variable

```
%let variable=1;
%macro routine;
   %put ***** Beginning ROUTINE *****;
   %local variable;
   %let variable=2;
   %put The value of variable inside ROUTINE is &variable;
   %put ***** Ending ROUTINE *****;
%mend routine;
%routine
%put The value of variable outside ROUTINE is &variable;
```

Submitting these statements writes these lines to the SAS log:

```
***** Beginning ROUTINE *****
The value of variable inside ROUTINE is 2
***** Ending ROUTINE *****
The value of variable outside ROUTINE is 1
```

%MACRO Macro Statement

Begins a macro definition.

Type:	Macro statement
Restriction:	Allowed in macro definitions or open code
See:	"%MEND Macro Statement" on page 413 and "SYSPBUFF Automatic Macro Variable" on page 265

Syntax

%MACRO *macro-name* <(*parameters*)> </ *options*>;

Required Arguments

macro-name
names the macro. A macro name must be a SAS name, which you supply; you cannot use a text expression to generate a macro name in a %MACRO statement. In addition, do not use macro reserved words as a macro name. (For a list of macro reserved words, see Appendix 1, " Reserved Words in the Macro Facility," on page 483.)

parameters
names one or more local macro variables whose values you specify when you invoke the macro. Parameters are local to the macro that defines them. You must supply each parameter name; you cannot use a text expression to generate it. A parameter list can contain any number of macro parameters separated by commas. The macro variables in the parameter list are usually referenced in the macro.

- *parameters* can be
  - <*positional-parameter-1*><,*positional-parameter-2* ...>
  - <*keyword-parameter*=<*value*> <,*keyword-parameter-2*=<*value*> ...>>

positional-parameter-1 <,*positional-parameter-2* ...>	specifies one or more positional parameters. You can specify positional parameters in any order, but in the macro invocation, the order in which you specify the values must match the order that you list them in the %MACRO statement. If you define more than one positional parameter, use a comma to separate the parameters. If at invocation that you do not supply a value for a positional parameter, the macro facility assigns a null value to that parameter.

keyword-parameter=<value> <,keyword-parameter-2=<value> ...>	names one or more macro parameters followed by equal signs. You can specify default values after the equal signs. If you omit a default value after an equal sign, the keyword parameter has a null value. Using default values enables you to write more flexible macro definitions and reduces the number of parameters that must be specified to invoke the macro. To override the default value, specify the macro variable name followed by an equal sign and the new value in the macro invocation.

Note: You can define an unlimited number of parameters. If both positional and keyword parameters appear in a macro definition, positional parameters must come first.

options

can be one or more of these optional arguments:

CMD

specifies that the macro can accept either a name-style invocation or a command-style invocation. Macros defined with the CMD option are sometimes called *command-style macros*.

Use the CMD option only for macros that you plan to execute from the command line of a SAS window. The SAS system option CMDMAC must be in effect to use command-style invocations. If CMDMAC is in effect and you have defined a command-style macro in your program, the macro processor scans the first word of every SAS command to see whether it is a command-style macro invocation. When the SAS system option NOCMDMAC option is in effect, the macro processor treats only the words following the % symbols as potential macro invocations. If the CMDMAC option is not in effect, you still can use a name-style invocation for a macro defined with the CMD option.

DES='*text*'

specifies a description for the macro entry in the macro catalog. The description text can be up to 256 characters in length. Enclose the description in quotation marks. This description appears in the CATALOG window when you display the contents of the catalog containing the stored compiled macros. The DES= option is especially useful when you use the stored compiled macro facility.

MINDELIMITER='*single character*';

specifies a value that will override the value of the MINDELIMITER= system option. The value must be a single character enclosed in single quotation marks and can appear only once in a %MACRO statement.

Restriction	The following characters cannot be used as a delimiter

`% & ' " ( ) ;`

Requirement	*Single character* must be enclosed in single quotation marks. If double quotation marks are used, an error results. This is different from system option MINDELIMITER=, which allows double or single quotation marks.
See	"MINDELIMITER= System Option" on page 455

MINOPERATOR
NOMINOPERATOR

specifies that the macro processor recognizes and evaluates the mnemonic **`IN`** and the special character **`#`** as logical operators when evaluating arithmetic or logical expressions during the execution of the macro. The setting of this argument overrides the setting of the NOMINOPERATOR global system option.

The NOMINOPERATOR argument specifies that the macro processor does not recognize the mnemonic **`IN`** and the special character **`#`** as logical operators when evaluating arithmetic or logical expressions during the execution of the macro. The setting of this argument overrides the setting of the MINOPERATOR global system option.

PARMBUFF

assigns the entire list of parameter values in a macro call, including the parentheses in a name-style invocation, as the value of the automatic macro variable SYSPBUFF. Using the PARMBUFF option, you can define a macro that accepts a varying number of parameter values.

If the macro definition includes both a set of parameters and the PARMBUFF option, the macro invocation causes the parameters to receive values. It also causes the entire invocation list of values to be assigned to SYSPBUFF.

To invoke a macro defined with the PARMBUFF option in a windowing environment or interactive line mode session without supplying a value list, enter an empty set of parentheses or more program statements after the invocation. This action indicates the absence of a value list, even if the macro definition contains no parameters.

SECURE
NOSECURE

causes the contents of a macro to be encrypted when stored in a stored compiled macro library.

The SECURE option enables you to write *secure* macros that prevent access to the macro source code, which helps to protect intellectual property contained in the code. For macros that are compiled with the SECURE option, the %COPY macro statement cannot recover the macro source code. When executing macros that are compiled with the SECURE option, system options MPRINT and MLOGIC are disabled so that the logical decisions made and the SAS code generated by the secure macro are no longer written to the SAS log.

However, due to the way that the macro facility processes and generates text, under certain conditions, it is still possible to recover the final SAS code generated by a secure macro. Secure macros should be used only to secure whole program segments, such as entire DATA and PROC steps and never for storing code fragments or passwords.

A NOSECURE option has been implemented to aid in the global edit of a source file or library to turn on security. For example, when you are creating several macros that will need to be secure. When creating the macros, use the NOSECURE option. When all macros are completed and ready for production, you can do a global edit and change NOSECURE to SECURE.

If you use the SECURE and SOURCE options on a macro, no output is produced when you use the %COPY statement. The following NOTE is written to the SAS log:

```
NOTE: The macro %name was compiled with the SECURE option. No
output will be produced for this %COPY statement.
```

STMT

specifies that the macro can accept either a name-style invocation or a statement-style invocation. Macros defined with the STMT option are sometimes called *statement-style macros*.

The IMPLMAC system option must be in effect to use statement-style macro invocations. If IMPLMAC is in effect and you have defined a statement-style macro in your program, the macro processor scans the first word of every SAS statement to see whether it is a statement-style macro invocation. When the NOIMPLMAC option is in effect, the macro processor treats only the words following the % symbols as potential macro invocations. If the IMPLMAC option is not in effect, you still can use a name-style invocation for a macro defined with the STMT option.

SOURCE
SRC

combines and stores the source of the compiled macro with the compiled macro code as an entry in a SAS catalog in a permanent SAS library. The SOURCE option requires that the STORE option and the MSTORED option be set. You can use the SASMSTORE= option to identify a permanent SAS library. You can store a macro or call a stored compiled macro only when the MSTORED option is in effect. (For more information, see "Storing and Reusing Macros" on page 137.)

Note: The source code saved by the SOURCE option begins with the %MACRO keyword and ends with the semi-colon following the %MEND statement.

CAUTION
The SOURCE option cannot be used on nested macro definitions (macro definitions contained within another macro).

STORE

stores the compiled macro as an entry in a SAS catalog in a permanent SAS library. Use the SAS system option SASMSTORE= to identify a permanent SAS library. You can store a macro or call a stored compiled macro only when the SAS system option MSTORED is in effect. (For more information, see “Storing and Reusing Macros” on page 137.)

Details

The %MACRO statement begins the definition of a macro, assigns the macro a name, and can include a list of macro parameters, a list of options, or both.

A macro definition must precede the invocation of that macro in your code. The %MACRO statement can appear anywhere in a SAS program, except within data lines. A macro definition cannot contain a CARDS statement, a DATALINES statement, a PARMCARDS statement, or data lines. Use an INFILE statement instead.

By default, a defined macro is an entry in a SAS catalog in the Work library. You can also store a macro in a permanent SAS catalog for future use. However, in SAS 6 and earlier, SAS does not support copying, renaming, or transporting macros.

You can nest macro definitions, but doing so is rarely necessary and is often inefficient. If you nest a macro definition, then it is compiled every time you invoke the macro that includes it. Instead, nesting a macro invocation inside another macro definition is sufficient in most cases.

Examples

Example 1: Using the %MACRO Statement with Positional Parameters

In this example, the macro PRNT generates a PROC PRINT step. The parameter in the first position is VAR, which represents the SAS variables that appear in the VAR statement. The parameter in the second position is SUM, which represents the SAS variables that appear in the SUM statement.

```
%macro prnt(var,sum);
   proc print data=srhigh;
      var &var;
      sum &sum;
   run;
%mend prnt;
```

In the macro invocation, all text up to the comma is the value of parameter VAR; text following the comma is the value of parameter SUM.

```
%prnt(school district enrollmt, enrollmt)
```

During execution, macro PRNT generates the following statements:

```
PROC PRINT DATA=SRHIGH;
   VAR SCHOOL DISTRICT ENROLLMT;
   SUM ENROLLMT;
RUN;
```

Example 2: Using the %MACRO Statement with Keyword Parameters

In the macro FINANCE, the %MACRO statement defines two keyword parameters, YVAR and XVAR, and uses the PLOT procedure to plot their values. Because the keyword parameters are usually EXPENSES and DIVISION, default values for YVAR and XVAR are supplied in the %MACRO statement.

```
%macro finance(yvar=expenses,xvar=division);
   proc plot data=yearend;
      plot &yvar*&xvar;
   run;
%mend finance;
```

- To use the default values, invoke the macro with no parameters.

  ```
  %finance()
  ```

  or

  ```
  %finance;
  ```

  The macro processor generates this SAS code:

  ```
  PROC PLOT DATA=YEAREND;
     PLOT EXPENSES*DIVISION;
  RUN;
  ```

- To assign a new value, give the name of the parameter, an equal sign, and the value:

  ```
  %finance(xvar=year)
  ```

  Because the value of YVAR did not change, it retains its default value. Macro execution produces this code:

  ```
  PROC PLOT DATA=YEAREND;
     PLOT EXPENSES*YEAR;
  RUN;
  ```

Example 3: Using the %MACRO Statement with the PARMBUFF Option

The macro PRINTZ uses the PARMBUFF option to enable you to input a different number of arguments each time you invoke it:

```
   %macro printz/parmbuff;
      %let num=1;
      %let color=%scan(&syspbuff,&num);
      %do %while(&color ne);
         %put Color &num.: %upcase(&color);
         %let num=%eval(&num+1);
         %let color=%scan(&syspbuff,&num);
      %end;
   %mend printz;
```

This invocation of PRINTZ contains four parameter values, `PURPLE`, `RED`, `BLUE`, and `TEAL` although the macro definition does not contain any individual parameters:

```
   %printz(purple,red,blue,teal)
```

As a result, SAS receives these statements:

```
%PUT COLOR 1: PURPLE;
%PUT COLOR 2: RED;
%PUT COLOR 3: BLUE;
%PUT COLOR 4: TEAL;
```

Example 4: Using the %MACRO Statement with the SOURCE Option

The SOURCE option combines and stores the source of the compiled macro with the compiled macro code. Use the %COPY statement to write the source to the SAS log. For more information about viewing or retrieving the stored source, see “%COPY Macro Statement” on page 385.

```
/* commentary */  %macro foobar(arg) /store source
      des="This macro does not do much";
%put arg = &arg;
* this is commentary!!!;
%* this is macro commentary;
%mend /* commentary; */;      /* Further commentary */

%copy foobar/source;
```

The following results are written to the SAS log:

```
%macro foobar(arg) /store source
des="This macro does not do much";
%put arg = &arg;
* this is commentary!!!;
%* this is macro commentary;
%mend /* commentary; */;
```

Example 5: Using the %MACRO Statement with the STORE and SECURE Options

The SECURE option can be used only in conjunction with the STORE option. The following example demonstrates the use of the STORE and an implied NOSECURE option to create a macro that is stored in plain text.

The following example demonstrates the use of the STORE and SECURE options to create a macro that is encrypted.

```
options mstored sasmstore=mylib;
libname mylib "SAS-library";
%macro secure/store secure; /* This macro is encrypted */
   data _null_;
      x=1;
      put "This data step was generated from a secure macro.";
   run;
%mend secure;

%secure

filename maccat catalog 'mylib.sasmacr.secure.macro';
data _null_;
   infile maccat;
```

```
      input;
      list;
   run;
```

Example 6: Adding a DATA Step within a Macro

You can create a DATA step within a macro.

```
%macro ds;
   data _null_;
      put "Fred";
   run;
%mend ds;

%ds;
```

```
Fred
NOTE: DATA statement used (Total process time):
      real time           0.05 seconds
      cpu time            0.02 seconds
```

%MEND Macro Statement

Ends a macro definition.

Type:	Macro statement
Restriction:	Allowed in macro definitions only

Syntax

%MEND <*macro-name*> ;

Required Argument

macro-name

names the macro as it ends a macro definition. Repeating the name of the macro is optional, but it is useful for clarity. If you specify *macro-name*, the name in the %MEND statement should match the name in the %MACRO statement; otherwise, SAS issues a warning message.

Example: Ending a Macro Definition

```
%macro prntData(dsn, obs=5);
   %if %sysfunc(exist(&dsn)) %then %do;
      proc print data=&dsn(obs=&obs);
      run;
   %end;
   %else %put ERROR: Data set &dsn not found.;
%mend prntData;
```

%PUT Macro Statement

Writes text or macro variable information to the SAS log.

Type:	Macro statement
Restriction:	Allowed in macro definitions or open code

Syntax

%PUT <*text* | _ALL_ | _AUTOMATIC_ | _GLOBAL_ | _LOCAL_ | _READONLY_ | _USER_ | _WRITABLE_>;

%PUT ERROR: *text* | NOTE: *text* | WARNING: *text*;

%PUT ERROR- *text* | NOTE- *text* | WARNING- *text*;

Without Arguments

When no arguments are specified, a blank line is written to the SAS log.

Optional Arguments

text

specifies text that is written to the SAS log. *Text* can be literal text or a text expresson. If *text* is longer than the current line size, the remainder of the text appears on the next line. The %PUT statement removes leading and trailing blanks from *text* unless you use a macro quoting function.

The %PUT statement is useful for debugging your macros. You can include macro variable references in the %PUT statement text to display their value at certain points during execution. As a shortcut, you can place &= before a macro variable name to print *macro-variable*=*value*. Here is an example:

```
%let x=1;
%put &=x;
```

The following is written to the SAS log:

```
X=1
```

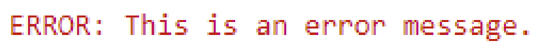
See "Using the %PUT Statement to Track Problems" on page 169.

ERROR: *text*

specifies an error message, which is displayed in red text in the SAS log. Here is an example.

```
%put ERROR: This is an error message.;
```

Here is how it appears in the SAS log.



```
ERROR: This is an error message.
```

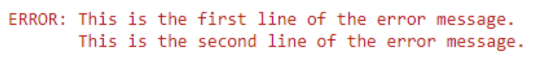
Requirement ERROR must be uppercase and followed immediately by a colon (:).

ERROR- ***text***

specifies an additional line of text for an error message.

```
%put ERROR: This is the first line of the error message.;
%put ERROR- This is the second line of the error message.;
```

Here is how it appears in the SAS log.



```
ERROR: This is the first line of the error message.
       This is the second line of the error message.
```

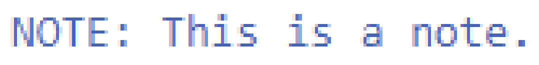
Requirement ERROR must be uppercase and followed immediately by a hyphen (-).

NOTE: ***text***

specifies a note, which is displayed in blue text in the SAS log. Here is an example.

```
%put NOTE: This is a note.;
```

Here is how it appears in the SAS log.



```
NOTE: This is a note.
```

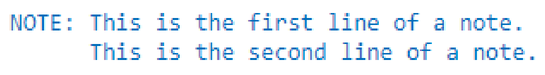
Requirement NOTE must be uppercase and followed immediately by a colon (:).

NOTE- ***text***

specifies an additional line of text for a note.

```
%put NOTE: This is the first line of a note.;
%put NOTE- This is the second line of a note.;
```

Here is how it appears in the SAS log.



```
NOTE: This is the first line of a note.
      This is the second line of a note.
```

Requirement NOTE must be uppercase and followed immediately by a hyphen (-).

Examples “Example 1: Using the SYSINCLUDEFILE Automatic Macro Variables in Included Code” on page 252

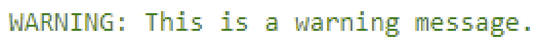
“Example 2: Using the SYSINCLUDEFILE Automatic Macro Variables in Included Macro Definitions” on page 253

WARNING: ***text***

specifies a warning message, which is displayed in green text in the SAS log. Here is an example.

```
%put WARNING: This is a warning message.;
```

Here is how it appears in the SAS log.



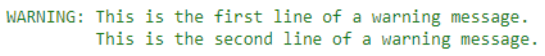
```
WARNING: This is a warning message.
```

Requirement WARNING must be uppercase and followed immediately by a colon (:).

WARNING- ***text***
specifies an additional line of text for a warning message.

```
%put WARNING: This is the first line of a warning message.;
%put WARNING- This is the second line of a warning message.;
```

Here is how it appears in the SAS log.



```
WARNING: This is the first line of a warning message.
         This is the second line of a warning message.
```

Requirement WARNING must be uppercase and followed immediately by a hyphen (-).

ALL
lists the values of all user-generated and automatic macro variables.

AUTOMATIC
lists the values of automatic macro variables. The automatic variables listed depend on the SAS products installed at your site and on your operating system. The scope is identified as AUTOMATIC.

GLOBAL
lists user-generated global macro variables. The scope is identified as GLOBAL.

LOCAL
lists user-generated local macro variables. The scope is the name of the currently executing macro.

READONLY
lists all user-defined read-only macro variables, regardless of scope. The scope is either GLOBAL, for global macro variables, or the name of the macro in which the macro variable is defined.

USER
lists user-generated global and local macro variables. The scope is identified either as GLOBAL, or as the name of the macro in which the macro variable is defined.

WRITABLE
lists all user-defined read and write macro variables, regardless of scope. The scope is either GLOBAL, for global macro variables, or the name of the macro in which the macro variable is defined.

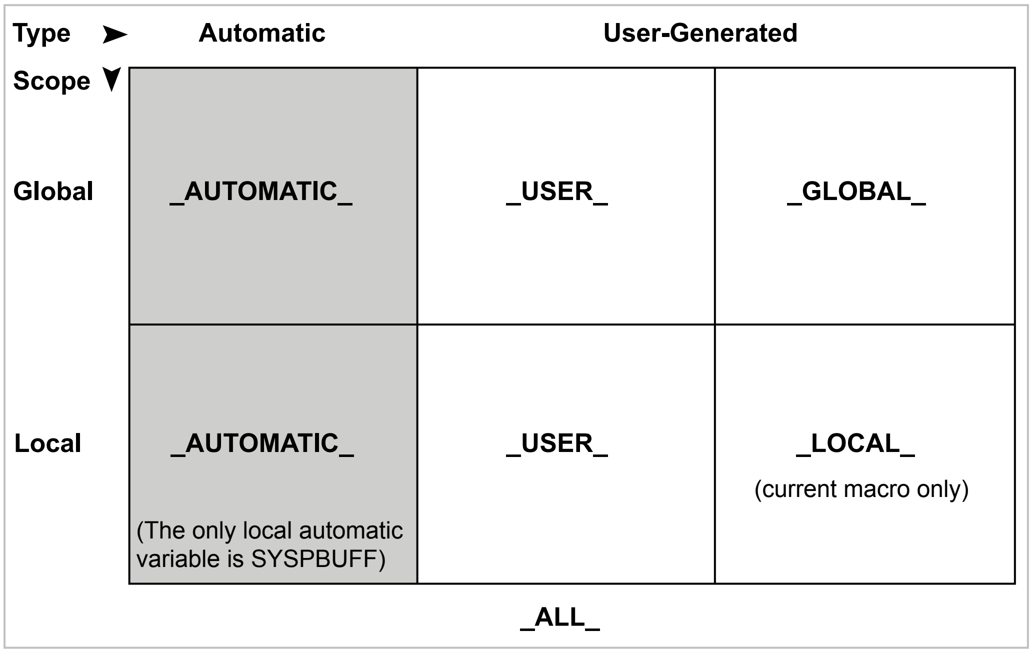
Details

When you use the %PUT statement to list macro variable descriptions, the %PUT statement includes only the macro variables that exist when statement executes. The description contains the macro variable's scope, name, and value. Macro variables with null values show only the scope and name of the variable. Characters in values that have been quoted with macro quoting functions remain quoted. Values that are too long for the current line size wrap to the next line or lines. Macro variables are listed in order from the current local macro variables outward to the global macro variables.

Note: The variables are always listed alphabetically. However, automatic macro variables created in a session might differ depending on which SAS products are installed and the platform on which SAS is running. Users are advised to refrain from writing code that depends on them appearing in a predictable order.

The following figure shows the relationship of these terms.

Figure 19.1 *%PUT Arguments by Type and Scope*



The %PUT statement displays text in different colors to generate messages that look like ERROR, NOTE, and WARNING messages generated by SAS. To display text in different colors, the first word in the %PUT statement must be ERROR, NOTE, or WARNING (all uppercase letters), followed immediately by a colon or a hyphen. You might also use the national-language equivalents of these words. Using a hyphen (-) following the ERROR, NOTE, or WARNING keyword causes the text of the %PUT statement to be a continuation of the previous ERROR, NOTE, or WARNING message, respectively. The specified text is indented to align with the previous lines of text, and is of the same color as the previous text.

Note: If you use the %PUT statement and the last message text that was generated by the SYSWARNINGTEXT and SYSERRORTEXT automatic macro variables contained an `&` or `%`, you must use the %SUPERQ macro quoting function. For more information, see "SYSERRORTEXT Automatic Macro Variable" on page 249 and "SYSWARNINGTEXT Automatic Macro Variable" on page 281.

Examples

Example 1: Displaying Text

The following statements illustrate using the %PUT statement to write text to the SAS log:

```
%put One line of text.;
%put %str(Use a semicolon(;) to end a SAS statement.);
%put %str(Enter the student%'s address.);
```

When you submit these statements, these lines appear in the SAS log:

```
One line of text.
Use a semicolon(;) to end a SAS statement.
Enter the student's address.
```

Example 2: Displaying Automatic Variables

To display all automatic variables, submit

```
%put _automatic_;
```

The result in the SAS log (depending on the products installed at your site) lists the scope, name, and value of each automatic variable.

Example 3: Displaying User-Generated Variables

This example lists the user-generated macro variables in all scopes.

```
%macro myprint(name);
   proc print data=&name;
      title "Listing of &name on &sysdate";
      footnote "&foot";
   run;
   %put _user_;
%mend myprint;
%let foot=Preliminary Data;
%myprint(consumer)
```

The %PUT statement writes these lines to the SAS log:

```
MYPRINT NAME consumer
GLOBAL FOOT Preliminary Data
```

Notice that SYSDATE does not appear because it is an automatic macro variable.

To display the user-generated variables after macro MYPRINT finishes, submit another %PUT statement.

```
%put _user_;
```

The result in the SAS log does not list the macro variable NAME because it was local to MYPRINT and ceased to exist when MYPRINT finished execution.

```
GLOBAL FOOT Preliminary Data
```

Example 4: Displaying Local Variables

This example displays the macro variables that are local to macro ANALYZE.

```
%macro analyze(name,vars,maxobsprint=10);
   proc freq data=&name;
      tables &vars;
   run;
   %put FIRST LIST:;
   %put _local_;
   %let firstvar=%scan(&vars,1);
   proc print data=&name(obs=&maxobsprint);
   run;
   %put SECOND LIST:;
   %put _local_;
%mend analyze;

%analyze(sashelp.cars,type cylinders origin)
```

In the result that is printed in the SAS log, the macro variable FIRSTVAR, which was created after the first %PUT _LOCAL_ statement, appears only in the second list.

```
FIRST LIST:
ANALYZE MAXOBSPRINT 10
ANALYZE NAME sashelp.cars
ANALYZE VARS type cylinders origin
SECOND LIST:
ANALYZE FIRSTVAR type
ANALYZE MAXOBSPRINT 10
ANALYZE NAME sashelp.cars
ANALYZE VARS type cylinders origin
```

%RETURN Macro Statement

Execution causes normal termination of the currently executing macro.

Type:	Macro Statement
Restriction:	Valid only in a macro definition

Syntax

%RETURN;

Details

The %RETURN macro causes normal termination of the currently executing macro.

Example: Using %RETURN Statement

In this example, if the error variable is set to 1, then the macro will stop executing and the DATA step will not execute.

```
%macro checkit(error);
   %if &error = 1 %then %return;
    data a;
       x=1;
    run;
%mend checkit;
%checkit(0)
%checkit(1)
```

%SYMDEL Macro Statement

Deletes the specified variable or variables from the macro global symbol table.

Type: Macro Statement

Syntax

%SYMDEL *macro-variable(s)</option>* ;

Required Arguments

macro-variable(s)
is the name of one or more macro variables or a text expression that generates one or more macro variable names. You cannot use a SAS variable list or a macro expression that generates a SAS variable list in a %SYMDEL statement.

option

NOWARN
suppresses the warning message when an attempt is made to delete a non-existent macro variable.

Details

%SYMDEL statement issues a warning when an attempt is made to delete a non-existent macro variable. To suppress this message, use the NOWARN option.

%SYSCALL Macro Statement

Invokes a SAS call routine.

Type:	Macro statement
Restriction:	Allowed in macro definitions or in open code
See:	"%SYSFUNC Macro Function" on page 356

Syntax

%SYSCALL *call-routine*< (*call-routine-argument(s)*)>;

Required Arguments

call-routine
: is a SAS or user-written CALL routine created with SAS/TOOLKIT software or a routine created using "FCMP Procedure" in *Base SAS Procedures Guide*. All SAS CALL routines are accessible with %SYSCALL except LABEL, VNAME, SYMPUT, and EXECUTE.

call-routine-argument(s)
: is one or more macro variable names (with no leading ampersands), separated by commas. You can use a text expression to generate part or all of the CALL routine arguments.

Details

When %SYSCALL invokes a CALL routine, the value of each macro variable argument is retrieved and passed unresolved to the CALL routine. Upon completion of the CALL routine, the value for each argument is written back to the respective macro variable. If %SYSCALL encounters an error condition, the execution of the CALL routine terminates without updating the macro variable values, an error message is written to the log, and macro processing continues.

Note: The arguments to %SYSCALL are evaluated according to the rules of the SAS macro language. This includes both the function name and the argument list to the function. In particular, an empty argument position will not generate a NULL argument, but a zero-length argument.

CAUTION

Do not use leading ampersands on macro variable names. The arguments in the CALL routine invoked by the %SYSCALL macro are resolved before execution. If you use leading ampersands, then the values of the macro variables are passed to the CALL routine rather than the names of the macro variables.

CAUTION

Macro variables contain only character data. When an argument to a function might be either numeric data or character data, %SYSCALL attempts to convert the supplied data to numeric data. If the value cannot be converted to numeric data, this value is passed in unchanged as character data.

When %SYSCALL is invoked, it evaluates each argument value and, if necessary, converts it to a data type that is required for the argument. For an argument that requires character data, %SYSCALL passes the argument value to the CALL routine as-is. For an argument that requires numeric data, %SYSCALL uses the %SYSEVALF on page 352 function to convert the character value or expression to a numeric value, and then passes the numeric value to the CALL routine. For an argument that can accept either numeric or character data, %SYSCALL silently attempts to convert the character value to a numeric value using the %SYSEVALF function. If the conversion is successful, it then passes the numeric value to the CALL routine. Otherwise, it passes the character value as-is to the CALL routine. To see whether the arguments are numeric only, character only, or either numeric or character, refer the documentation for the desired CALL routine in "Dictionary of Functions and CALL Routines" in *SAS Functions and CALL Routines: Reference*.

The way in which %SYSCALL evaluates argument values that can be numeric or character can cause unexpected results in some cases. For example, when a character value includes trailing blanks, the trailing blanks are removed when %SYSCALL attempts to convert the value to numeric data. If the trailing blanks are significant, the result is undesirable. When an argument value that is not intended as an expression includes mathematical symbols or mnemonic operators, %SYSCALL might convert the argument value to numeric data. The result can be undesirable. Here is a simple demonstration of these results.

```
%let p1 = %str(A    );
%let p2 = BC;
%let p3 = EQ;
%let count = 1;
%syscall allperm(count, p1, p2, p3);
%put &p1&p2&p3;
```

Here is the expected result:

```
A    BCEQ
```

Here is the actual result:

```
ABC1
```

The ALLPERM CALL routine accepts numeric data for the first argument, and numeric or character data for the subsequent arguments. Notice the following:

- The trailing blanks are removed from the p1 value. This is not the desired result.
- The p2 value is unchanged. This is the desired result.

- The p3 value is replaced with 1. This is not the desired result. Because EQ is the equals mnemonic operator, when it is evaluated, %SYSEVALF(EQ) returns 1 (null EQ null is True). %SYSCALL then passes 1 to the ALLPERM CALL routine for this argument instead of EQ.

To resolve these issues, use the %QUOTE on page 334 function when assigning the value to p1, and use the %STR on page 342 function when assigning the value to p3 as follows:

```
%let p1 = %quote(A    );
%let p2 = BC;
%let p3 = %str(EQ);
%let count = 1;
%syscall allperm(count, p1, p2, p3);
%put &p1&p2&p3;
```

The %QUOTE function preserves the trailing blanks in the p1 value, and the %STR function masks mnemonic operator EQ in the p3 value. This produces the desired result:

```
A    BCEQ
```

Examples

Example 1: Using the RANUNI Call Routine with %SYSCALL

This example illustrates how to use a %SYSCALL statement to invoke the SAS CALL routine RANUNI.

Note: The syntax for RANUNI is **RANUNI(seed,x)**.

```
%let a = 123456;
%let b = 0;
%syscall ranuni(a,b);
%put &a, &b;
```

The %PUT statement writes the following values of the macro variables A and B to the SAS log:

```
1587033266 0.739019954
```

Example 2: Using the CATX Call Routine with %SYSCALL

This example illustrates how to use a %SYSCALL statement to invoke the SAS CALL routine CATX.

Create the input macro variables. The macro variables contain the argument data. Because the CATX routine accepts numeric or character data for the third and subsequent arguments, use the %STR on page 342 function to mask the NE mnemonic operator in s3. This prevents the s3 value from being converted to numeric data.

```
%let s1 = Wind:;
```

```
%let s2 = %str(NE);
%let s3 = 15-25 MPH;
%let delim = %str( );
```

Create the result macro variable.

```
data _null_;
   length msg $40;
   call symput('msg',msg);
run;
```

Use a %SYSCALL macro statement to invoke the SAS CALL routine CATX.

```
%syscall catx(delim, msg, s1, s2, s3);
%put %sysfunc(trim(&msg));
```

Here is the result.

```
Wind: NE 15-25 MPH
```

%SYSEXEC Macro Statement

Executes Linux commands.

Type:	Macro statement
Requirement:	XCMD must be enabled on the compute server to run the %SYSEXEC statement. By default, XCMD is disabled.
Note:	You can use the %SYSEXEC statement inside a macro or in open code.
See:	"SYSSCP and SYSSCPL Automatic Macro Variable" on page 270 and "SYSRC Automatic Macro Variable" on page 269

Syntax

%SYSEXEC <*command*> ;

Required Arguments

no argument
starts a new Linux shell, where you can issue Linux commands and return to your Compute Server session.

command
is any Linux command. If *command* contains a semicolon, use a macro quoting function.

Details

The %SYSEXEC statement causes the operating environment to immediately execute the command that you specify and assigns any return code from the operating environment to the automatic macro variable SYSRC. Use the %SYSEXEC statement with automatic macro variables SYSSCP and SYSSCPL to write portable macros that run under multiple operating environments.

Comparisons

The %SYSEXEC statement is analogous to the X command. However, unlike the X command, commands invoked with %SYSEXEC should not be enclosed in quotation marks.

Example

The following code writes the status of the default printer to your Linux shell:

```
%sysexec lpstat;
```

%SYSLPUT Macro Statement

Creates a new macro variable or modifies the value of an existing macro variable on a remote host or server.

Type:	Macro Statement
Restriction:	Allowed in macro definitions or open code
Requirement:	SAS/CONNECT
See:	"Use the Macro Facility with SAS/CONNECT" in *SAS/CONNECT User's Guide* "%LET Macro Statement" on page 402 and "%SYSRPUT Macro Statement" on page 429

Syntax

%SYSLPUT *macro-variable*=<*value*></ REMOTE=*server-ID*>;

%SYSLPUT *macro-variable-type* </ *option(s)*;>;

Required Arguments

macro-variable=<value>

specifies either the name of a macro variable followed by = and an optional value or a macro expression that produces a macro variable name followed by = and an optional value. The name can refer to a new or existing macro variable on a remote host or server. It must be followed by = even if a value is not specified.

value

is a macro variable reference, a macro invocation, or the character value to be assigned to the server *macro-variable*.

Omitting the value produces a null (0 characters). Leading and trailing blanks are ignored. To make them significant, enclose the value in the %STR function. The character value should not contain nested quotation marks.

Requirement	Values containing special characters, such as the forward slash (/) or single quotation mark (‘), must be masked using the %BQUOTE function so that the macro processor correctly interprets the special character as part of the text and not as an element of the macro language. See “Mask Character Values with %BQUOTE (Form 1)” in *SAS/CONNECT User’s Guide* for an example of how to use the %BQUOTE function. For more information about Macro Quoting in general, see Chapter 7, “Macro Quoting,” on page 95.

Valid in	Form 1 only

macro-variable-type

is the type of macro variable to copy. Can be one of the following:

ALL

copies all user-generated and automatic macro variables to the server session.

AUTOMATIC

copies all automatic macro variables to the server session.

The automatic variables copied depend on the SAS products installed at your site and on your operating system. The scope is identified as AUTOMATIC.

GLOBAL

copies all user-generated global macro variables to the server session.

The scope is identified as GLOBAL.

LOCAL

copies all user-generated local macro variables to the server session.

The scope is the name of the currently executing macro.

USER

pushes all user-defined macro variables from the local host to the remote host or server all at the same time.

Valid in	Form 2 only

Optional Arguments

REMOTE=*server-ID*

specfies the name of the server session that the macro variable will be created in. If only one server session is active, the *server-ID* can be omitted. If multiple server sessions are active, omitting this option causes the macro to be created in the most recently accessed server session. You can find out which server session is currently active by examining the value that is assigned to the CONNECTREMOTE system option.

Interactions	The REMOTE= option that is specified in the %SYSLPUT macro statement overrides the CONNECTREMOTE= system option.
	The REMOTE= and LIKE= options are independent of each other and can be specified on the same %SYSLPUT statement, regardless of order.

LIKE='*character-string*'

specifies a sequence of characters, or pattern, to be used as the criteria for determining which macro variables are to be copied to the server session. Character patterns can consist of the following:

- any sequence of characters, A-Z
- any sequence of digits, 0-9
- a single wildcard character in the form of an asterisk (*)

The wildcard character (*) cannot be embedded or used more than once in the character string.

Valid in	Form 2 only
Restrictions	The wildcard character (*) cannot be embedded in the character string.
	The wildcard character (*) can be specified only once in the character string.
Requirement	The wildcard character (*) must be used at either the beginning or the end of the character string.
Note	The LIKE= option is not case sensitive.
See	For more detailed information, see "/LIKE=<'character-string' >" in *SAS/CONNECT User's Guide*.

Details

The %SYSLPUT statement is submitted with SAS/CONNECT software from the local host or client to a remote host or server to create a new macro variable on the remote host or server, or to modify the value of an existing macro variable on the remote host or server.

Note: The names of the macro variables on the remote and local hosts must not contain any leading ampersands.

To assign the value of a macro variable on a remote host to a macro variable on the local host, use the %SYSRPUT statement.

To use %SYSLPUT, you must have initiated a link between a local Compute Server session or client and a remote Compute Server session or server using the SIGNON command or SIGNON statement. For more information, see the documentation for SAS/CONNECT software.

%SYSMACDELETE Macro Statement

Deletes a macro definition from the Work.SASMacr or Work.SASMac*n* catalog.

Type:	Macro Statement
Restriction:	Allowed in macro definition and open code
Note:	The Work.Sasmacr catalog is used to store compiled macros for the primary SAS session. In applications or programs that use side sessions, the catalog used to store compiled macros for each side session is Work.Sasmac*n*, where *n* is a unique integer.

Syntax

%SYSMACDELETE *macro_name* </ *option*>;

Required Argument

macro_name
the name of a macro or a text expression that produces a macro variable name.

Optional Argument

NOWARN
specifies that no warning diagnostic message should be issued.

Details

The %SYSMACDELETE statement deletes the macro definition of the specified macro from the Work.SASMacr or Work.SASMac*n* catalog. If no definition for the macro exists in the Work.SASMacr or Work.SASMac*n* catalog, a WARNING diagnostic message is issued. If the macro is currently being executed, an ERROR diagnostic message is issued.

%SYSMSTORECLEAR Macro Statement

Closes the stored compiled macro catalog associated with the libref specified in the SASMSTORE= option.

Type:	Macro statement
Restriction:	Allowed in macro definition and open code
Note:	This statement does not clear the libref specified in the SASMSTORE= option.
See:	SASMSTORE= system option

Syntax

%SYSMSTORECLEAR;

Details

Use the %SYSMSTORECLEAR statement to close the stored compiled macro catalog so that you can clear the previous libref when switching between SASMSTORE= libraries.

Note: If any stored compiled macro from the library specified by the SASMSTORE= system option is still executing, an ERROR diagnostic message is issued, and the library is not closed.

%SYSRPUT Macro Statement

Assigns a value from the remote host to a macro variable on the local host.

Type:	Macro statement
Restriction:	Allowed in macro definitions or open code
Requirement:	SAS/CONNECT
See:	"Use the Macro Facility with SAS/CONNECT" in *SAS/CONNECT User's Guide* "SYSERR Automatic Macro Variable" on page 247, "SYSINFO Automatic Macro Variable" on page 256, and "%SYSLPUT Macro Statement" on page 425

Syntax

%SYSRPUT *local-macro-variable=remote-macro-variable* | *value*;

%SYSRPUT _USER_ < / LIKE='*character-string*'> ;

Required Arguments

local-macro-variable

is the name of a macro variable with no leading ampersand or a text expression that produces the name of a macro variable. This name must be a macro variable that is stored on the local host.

Valid in	Form 1 only

remote-macro-variable

is the name of a macro variable with the leading ampersand or a text expression that produces the name of a macro variable with the leading ampersand. This name must be a macro variable that is stored on a remote host.

Valid in	Form 1 only

value

is a value or an expression that produces a value.

Valid in	Form 1 only

USER

to push all remote user-defined macro variables from the remote host to the local host all at the same time.

Valid in	Form 2 only

Optional Argument

LIKE='*character-string*'

Specifies the sequence of characters, or pattern, to be used as the criteria for determining which macro variables are to be copied to the server session. Character patterns can consist of the following:

- any sequence of characters, A–Z
- any sequence of digits, 0–9
- a single wildcard character in the form of an asterisk (*)

The wildcard character (*) cannot be embedded or used more than once in the character string.

Valid in	Form 2 only
Restrictions	The wildcard character (*) cannot be embedded in the character string.

	The wildcard character (*) can be specified only once in the character string.
Requirement	The wildcard character (*) must be used at either the beginning or the end of the character string.
See	For more detailed information, see "/LIKE=<'character-string' >" in *SAS/CONNECT User's Guide*.

Details

The %SYSRPUT statement is submitted with SAS/CONNECT to a remote host to retrieve a value that is 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 into the global symbol table in the client session.

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 Compute Server 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.

For more information about using %SYSRPUT, see the documentation for SAS/CONNECT Software.

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

Example: Checking 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 from a noninteractive job. When remote processing is completed, the job then checks the value of the return code stored in RETCODE. Processing continues on the local host if the remote processing is successful.

The %SYSRPUT statement is useful for capturing the value returned in the SYSINFO macro variable and passing that value to the local host. The SYSINFO macro variable contains return-code information provided by SAS procedures. In the example, the %SYSRPUT statement follows a PROC DOWNLOAD step, so the value returned by SYSINFO indicates the success of the PROC DOWNLOAD step:

```
rsubmit;
   %macro download;
      proc download data=remote.mydata out=local.mydata;
      run;
      %sysrput retcode=&sysinfo;
   %mend download;
   %download
endrsubmit;
%macro checkit;
   %if &retcode = 0 %then %do;
      further processing on local host
   %end;
%mend checkit;
%checkit
```

A SAS/CONNECT batch (noninteractive) job always returns a system condition code of 0. To determine the success or failure of the SAS/CONNECT noninteractive job, use the %SYSRPUT macro statement to check the value of the automatic macro variable SYSERR. To determine what remote system the SAS/CONNECT conversation is attached to, remote submit the following statement:

```
%sysrput rhost=&sysscp;
```

20

System Options for Macros

System Options for Macros

There are several SAS system options that apply to the macro facility.

Dictionary

CMDMAC System Option

Controls command-style macro invocation.

Valid in:	Configuration file, OPTIONS window, OPTIONS statement, SAS invocation
Category:	Macro
PROC OPTIONS GROUP=	MACRO
Type:	System option
Default:	NOCMDMAC

Syntax

CMDMAC

NOCMDMAC

Syntax Description

CMDMAC
specifies that the macro processor examine the first word of every windowing environment command to see whether it is a command-style macro invocation.

Note: When CMDMAC is in effect, SAS searches the macro libraries first and executes any member that it finds with the same name as the first word in the windowing environment command that was issued. Unexpected results can occur.

NOCMDMAC
: specifies that no check be made for command-style macro invocations. If the macro processor encounters a command-style macro call when NOCMDMAC is in effect, it treats the call as a SAS command and produces an error message if the command is not valid or is not used correctly.

Details

The CMDMAC system option controls whether macros defined as command-style macros can be invoked with command-style macro calls or if these macros must be invoked with name-style macro calls. These two examples illustrate command-style and name-style macro calls, respectively:

- `macro-name parameter-value-1 parameter-value-2`
- `%macro-name(parameter-value-1, parameter-value-2)`

When you use CMDMAC, processing time is increased because the macro facility searches the macros compiled during the current session for a name corresponding to the first word on the command line. If the MSTORED option is in effect, the libraries containing compiled stored macros are searched for a name corresponding to that word. If the MAUTOSOURCE option is in effect, the autocall libraries are searched for a name corresponding to that word. If the MRECALL system option is also in effect, processing time can be increased further because the search continues even if a word was not found in a previous search.

Regardless of which option is in effect, you can use a name-style invocation to call any macro, including command-style macros.

Comparisons

Name-style macros are the more efficient choice for invoking macros because the macro processor searches only for a macro name corresponding to a word following a percent sign.

IMPLMAC System Option

Controls statement-style macro invocation.

Valid in:	Configuration file, OPTIONS window, OPTIONS statement, SAS invocation
Category:	Macro
PROC OPTIONS GROUP=	MACRO
Type:	System option
Default:	NOIMPLMAC

Syntax

IMPLMAC

NOIMPLMAC

Syntax Description

IMPLMAC
specifies that the macro processor examine the first word of every submitted statement to see whether it is a statement-style macro invocation.

Note: When IMPLMAC is in effect, SAS searches the macro libraries first and executes any macro that it finds with the same name as the first word in the SAS statement that was submitted. Unexpected results can occur.

NOIMPLMAC
specifies that no check be made for statement-style macro invocations. This is the default. If the macro processor encounters a statement-style macro call when NOIMPLMAC is in effect, it treats the call as a SAS statement. SAS produces an error message if the statement is not valid or if it is not used correctly.

Details

The IMPLMAC system option controls whether macros defined as statement-style macros can be invoked with statement-style macro calls or if these macros must be invoked with name-style macro calls. These examples illustrate statement-style and name-style macro calls, respectively:

- `macro-name parameter-value-1  parameter-value-2;`
- `%macro-name(parameter-value-1, parameter-value-2)`

When you use IMPLMAC, processing time is increased because SAS searches the macros compiled during the current session for a name corresponding to the first word of each SAS statement. If the MSTORED option is in effect, the libraries containing compiled stored macros are searched for a name corresponding to that word. If the MAUTOSOURCE option is in effect, the autocall libraries are searched for a name corresponding to that word. If the MRECALL system option is also in effect, processing time can be increased further because the search continues even if a word was not found in a previous search.

Regardless of which option is in effect, you can call any macro with a name-style invocation, including statement-style macros.

Note: If a member in an autocall library or stored compiled macro catalog has the same name as an existing windowing environment command, SAS searches for the macro first if CMDMAC is in effect. Unexpected results can occur.

Comparisons

Name-style macros are a more efficient choice to use when you invoke macros because the macro processor searches only for the macro name that corresponds to a word that follows a percent sign.

MACRO System Option

Controls whether the SAS macro language is available.

Valid in:	Configuration file, SAS invocation
Category:	Macro
PROC OPTIONS GROUP=	MACRO
Type:	System option
Default:	MACRO

Syntax

MACRO

NOMACRO

Syntax Description

MACRO
enables SAS to recognize and process macro language statements, macro calls, and macro variable references.

NOMACRO
prevents SAS from recognizing and processing macro language statements, macro calls, and macro variable references. The item generally is not recognized, and an error message is issued. If the macro facility is not used in a job, a small performance gain can be made by setting NOMACRO because there is no overhead of checking for macros or macro variables.

MACROCOMPVERWARN System Option

Specifies whether a warning or a note is written to the SAS log when a stored compiled macro that was compiled on a different operating system or a different version of SAS is executed.

Valid in:	Configuration file, OPTIONS window, OPTIONS statement, SAS invocation
Category:	Macro

PROC OPTIONS GROUP=	MACRO
Type:	System option
Default:	MACROCOMPVERWARN
Note:	This option is valid starting with 2022.1.4.
See:	SAS log

Syntax

MACROCOMPVERWARN

NOMACROCOMPVERWARN

Syntax Description

MACROCOMPVERWARN

causes a WARNING message to be written to the SAS log when a stored compiled macro that was compiled on a different operating system or a different version of SAS is executed. Here is an example of the warning message text:

```
WARNING: The macro macro-name was compiled with the SAS System V9.4.
 The current version of the SAS System is V.04.00. The macro might not execute
 correctly. To avoid this message, recompile the macro with the V.04.00 version
 of the SAS System.
```

The warning message causes the SAS job to return a status code of 1 instead of 0.

NOMACROCOMPVERWARN

causes a NOTE to be written to the SAS log when a stored compiled macro that was compiled on a different operating system or a different version of SAS is executed. Here is an example of the note text:

```
NOTE: The macro macro-name was compiled with the SAS System V9.4.
 The current version of the SAS System is V.04.00. The macro might not execute
 correctly. To avoid this message, recompile the macro with the V.04.00 version
 of the SAS System.
```

The note does not cause the SAS job to return a status code of 1.

Details

By default, a warning message results when a stored compiled macro that was compiled on a different operating system or a different version of SAS is executed. Starting with 2022.1.4, you can use the NOMACROCOMPVERWARN system option to write a note to the SAS log instead of a warning in this case. For example, assume that macro %HELLOWORLD was compiled and stored in SAS 9.4 using the following code:

```
filename macstore "compiled-macro-store-path";
```

```
options mstored sasmstore=macstore;
%macro helloWorld / store source
     des='Hello World';
   %put;
   %put Hello, World!;
%mend helloWorld;
```

To run this macro without a warning in 2022.1.4 and later releases, use the following code:

```
filename macstore "compiled-macro-store-path";
options mstored sasmstore=macstore nomacrocompverwarn;
%helloWorld;
options macrocompverwarn source;
```

Here is the result:

```
NOTE: The macro HELLOWORLD was compiled with the SAS System V9.4.
 The current version of the SAS System is V.04.00. The macro might not execute
 correctly. To avoid this message, recompile the macro with the V.04.00 version
 of the SAS System.

Hello, World!
```

The note identifies the issue, but it does not cause the SAS job to return a status code of 1.

Executing stored compiled macros that were compiled on a different operating system or a different version of SAS is not supported. Unexpected issues can occur in that case. If you receive these warnings or notes when you execute your stored compiled macros, recompile your stored compiled macros in the current release of SAS, if possible.See "Saving Macros Using the Stored Compiled Macro Facility " on page 141.

MAUTOCOMPLOC System Option

Displays in the SAS log the source location of an autocall macro when the autocall macro is compiled.

Valid in:	Configuration file, OPTIONS window, OPTIONS statement, SAS invocation
Category:	Macro
PROC OPTIONS GROUP=	MACRO
Type:	System option
Default:	NOMAUTOCOMPLOC

Syntax

MAUTOCOMPLOC

NOMAUTOCOMPLOC

Syntax Description

MAUTOCOMPLOC
displays the autocall macro source location in the SAS log when the autocall macro is compiled.

NOMAUTOCOMPLOC
prevents the autocall macro source location from being written to the SAS log.

Details

The display created by the MAUTOCOMPLOC system option of the autocall macro source location in the log is not affected by either the MAUTOLOCDISPLAY or the MLOGIC system options.

MAUTOLOCDISPLAY System Option

Specifies whether to display the source location of the autocall macros in the log when the autocall macro is invoked.

Valid in:	Configuration file, OPTIONS window, OPTIONS statement, SAS invocation
Category:	Macro
PROC OPTIONS GROUP=	MACRO
Type:	System option
Default:	NOMAUTOLOCDISPLAY

Syntax

MAUTOLOCDISPLAYNOMAUTOLOCDISPLAY

NOMAUTOLOCDISPLAY

Syntax Description

MAUTOLOCDISPLAY
enables MACRO to display the autocall macro source location in the log when the autocall macro is invoked.

NOMAUTOLOCDISPLAY
prevents the autocall macro source location from being displayed in the log when the autocall macro is invoked. NOMAUTOLOCDISPLAY is the default.

Details

When both MAUTOLOCDISPLAY and MLOGIC options are set, only the MLOGIC listing of the autocall source location is displayed.

MAUTOLOCINDES System Option

Specifies whether the macro processor prepends the full pathname of the autocall source file to the description field of the catalog entry of compiled autocall macro definition in the Work.SASMacr or Work.SASMac*n* catalog.

Valid in:	Configuration file, OPTIONS window, OPTIONS statement, SAS invocation
Category:	Macro
PROC OPTIONS GROUP=	MACRO
Type:	System option
Default:	NOMAUTOLOCINDES
Note:	The Work.Sasmacr catalog is used to store compiled macros for the primary SAS session. In applications or programs that use side sessions, the catalog used to store compiled macros for each side session is Work.Sasmac*n*, where *n* is a unique integer.
See:	SAS log

Syntax

MAUTOLOCINDES

NOMAUTOLOCINDES

Syntax Description

MAUTOLOCINDES
causes the macro processor to prepend the full pathname of the autocall macro source file to the description field of the catalog entry of the compiled autocall macro definition in the Work.SASMacr or Work.SASMac*n* catalog.

NOMAUTOLOCINDES
no changes to the description field autocall macro definitions in the Work.SASMacr or Work.SASMac*n* catalog.

Details

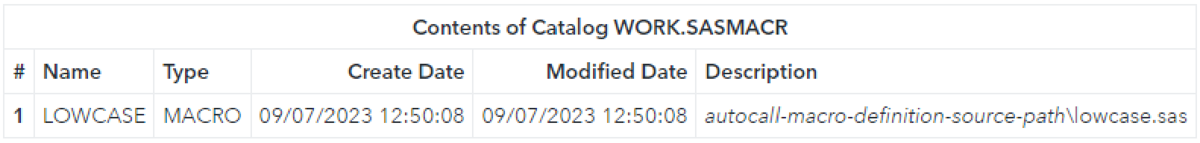
Use MAUTOLOCINDES to help determine where autocall macro definition source code is located. The following is an example that shows the output that contains the full pathname:

```
options mautolocindes;
%put %lowcase(THIS);

proc catalog cat=work.sasmacr;
   contents;
run;
quit;
options nomautolocindes;
```

The following is written to the SAS log:

```
this
```

Here is an example of the output from the CATALOG procedure. The DESCRIPTION column shows the full path to the source for each macro.



Contents of Catalog WORK.SASMACR

#	Name	Type	Create Date	Modified Date	Description
1	LOWCASE	MACRO	09/07/2023 12:50:08	09/07/2023 12:50:08	*autocall-macro-definition-source-path*\lowcase.sas

MAUTOSOURCE System Option

Specifies whether the autocall feature is available.

Valid in:	Configuration file, OPTIONS window, OPTIONS statement, SAS invocation
Category:	Macro
PROC OPTIONS GROUP=	MACRO
Type:	System option
Default:	MAUTOSOURCE

Syntax

MAUTOSOURCE

NOMAUTOSOURCE

Syntax Description

MAUTOSOURCE
causes the macro processor to search the autocall libraries for a member with the requested name when a macro name is not found in the Work library.

NOMAUTOSOURCE
prevents the macro processor from searching the autocall libraries when a macro name is not found in the Work library.

Details

When the macro facility searches for macros, it searches first for macros compiled in the current Compute Server session. If the MSTORED option is in effect, the macro facility next searches the libraries containing compiled stored macros. If the MAUTOSOURCE option is in effect, the macro facility next searches the autocall libraries.

MCOMPILE System Option

Specifies whether to allow new definitions of macros.

Valid in:	Configuration file, OPTIONS window, OPTIONS statement, SAS invocation
Category:	Macro
PROC OPTIONS GROUP=	MACRO
Type:	System option
Default:	MCOMPILE

Syntax

MCOMPILE

NOMCOMPILE

Syntax Description

MCOMPILE
allows new macro definitions.

NOMCOMPILE
disallows new macro definitions.

Details

The MCOMPILE system option allows new definitions of macros.

The NOMCOMPILE system option prevents new definitions of macros. It does not prevent the use of existing stored compiled or autocall macros.

MCOMPILENOTE= System Option

Issues a NOTE to the SAS log. The note contains the size and number of instructions upon the completion of the compilation of a macro.

Valid in:	Configuration file, OPTIONS window, OPTIONS statement, SAS invocation
Category:	Macro
PROC OPTIONS GROUP=	MACRO
Type:	System option
Default:	NONE

Syntax

MCOMPILENOTE=NONE | NOAUTOCALL | ALL

Required Arguments

NONE
prevents any NOTE from being written to the log.

NOAUTOCALL
prevents any NOTE from being written to the log for AUTOCALL macros, but does issue a NOTE to the log upon the completion of the compilation of any other macro.

ALL
issues a NOTE to the log. The note contains the size and number of instructions upon the completion of the compilation of any macro.

Details

The NOTE confirms that the compilation of the macro was completed. When the option is on and the NOTE is issued, the compiled version of the macro is available for execution. A macro can successfully compile, but still contain errors or warnings that will cause the macro to not execute as you intended.

Example: Using MCOMPILENOTE System Option

A macro can actually compile and still contain errors. Here is an example of the NOTE without errors:

```
option mcompilenote=noautocall;
%macro mymacro;
%mend mymacro;
```

Output to the log:

```
NOTE: The macro MYMACRO completed compilation without errors.
```

Here is an example of the NOTE with errors:

```
%macro yourmacro;
%end;
%mend yourmacro;
```

Output to the log:

```
ERROR: There is no matching %DO statement for the %END statement.
       This statement will be ignored.
NOTE: The macro YOURMACRO completed compilation with errors.
```

MCOVERAGE System Option

Enables the generation of coverage analysis data.

Valid in:	Configuration file, OPTIONS window, OPTIONS statement, SAS invocation
Category:	Macro
PROC OPTIONS GROUP=	Macro
Type:	System option
Default:	NOMCOVERAGE
Requirement:	Must use MCOVERAGELOC= system option

Syntax

MCOVERAGE

NOMCOVERAGE

Syntax Description

MCOVERAGE
enables the generation of coverage analysis data.

NOMCOVERAGE
prevents the generation of coverage analysis data.

Details

MCOVERAGE system option controls the generation of *coverage analysis data*, which is information needed to ensure proper testing of SAS Solutions products before their release.

The format of the coverage analysis data is a space delimited flat text file that contains three types of records. Each record begins with a numeric record type. The line numbers in the data are relative line numbers based on the %MACRO keyword used to define the macro. You must use the MCOVERAGELOC= system option to specify the location of the coverage analysis data file. See “MCOVERAGELOC= System Option” on page 449.

Note: Because nested macro definitions are stored as model text with line breaks collapsed, it is recommended that nested macro definitions not be used in macro definitions that will later be analyzed for execution coverage.

Below are explanations for each of the three record types.

Record type 1:

`1 n n macroname`

1
: record type

`n`
: first line number

`n`
: last line number

`macroname`
: macro name

Record type 1 indicates the beginning of the execution of a macro. Record type 1 appears once for each invocation of a macro.

Record type 2:

`2 n n macroname`

2
: record type

`n`
: first line number

`n`
: last line number

`macroname`
: macro name

Record type 2 indicates the lines of a macro that have executed. A single line of a macro might cause more than one record to be generated.

Record type 3:

`3 n n macroname`

3
: record type

`n`
: first line number

`n`
: last line number

`macroname`
: macro name

Record type 3 indicates which lines of the macro cannot be executed because no code was generated from them. These lines might be either commentary lines or lines that cause no macro code to be generated.

The following is a sample program log:

```
Sample Program Log:
NOTE: Copyright (c) 2002-2012 by SAS Institute Inc., Cary, NC, USA.
NOTE: SAS (r) Proprietary Software 9.4 (TS1B0)
      Licensed to SAS Institute Inc., Site 1.
NOTE: This session is executing on the W32_7PRO  platform.



NOTE: SAS initialization used:
      real time           0.45 seconds
      cpu time            0.20 seconds

1          options source source2;
2
3          options mcoverage mcoverageloc='./foo.dat';
4
5          /*  1 */ %macro
6          /*  2 */ foo (
7          /*  3 */ arg,
8          /*  4 */
9          /*  5 */
10         /*  6 */ arg2
11         /*  7 */
12         /*  8 */
13         /*  9 */ =
14         /* 10 */
15         /* 11 */ This is the default value of arg2)
16         /* 12 */ ;
17         /* 13 */ /*  This is a number of lines of comments     */
18         /* 14 */ /*  which presumably will help the maintainer */
19         /* 15 */ /*  of this macro to know what to do to keep  */
20         /* 16 */ /*  this silly piece of code current          */
21         /* 17 */   %if &arg %then %do;
22         /* 18 */      data _null_;
23         /* 19 */      x=1;
24         /* 20 */   %end;
25         /* 21 */ %* this is a macro comment statement
26         /* 22 */    that also can be used to document features
27         /* 23 */    and other stuff about the macro;
28         /* 24 */   %else
29         /* 25 */   %do;
30         /* 26 */      DATA _NULL_;
31         /* 27 */      y=1;
32         /* 28 */   %end;
33         /* 29 */      run;
34         /* 30 */
35         /* 31 */
36         /* 32 */
37         /* 33 */
38         /* 34 */
39         /* 35 */ %mend
40         /* 36 */
41         /* 37 */
42         /* 38 */
43         /* 39 */
44         /* 40 */
45         /* 41 */
46         /* 42 */
47         /* 43 */ foo This is text which should generate a warning!  ;
WARNING: Extraneous information on %MEND statement ignored for macro
definition FOO.
```

MCOVERAGELOC= System Option

Specifies the location of the coverage analysis data file.

Valid in:	Configuration file, OPTIONS window, OPTIONS statement, SAS invocation
Category:	Macro
PROC OPTIONS GROUP=	Macro
Type:	System option
Requirement:	Use with MCOVERAGE system option
See:	"MCOVERAGE System Option" on page 445

Syntax

MCOVERAGELOC=*fileref* | *file-specification*

Required Argument

fileref | file-specification
a SAS fileref or an external file specification enclosed in quotation marks.

Details

This MCOVERAGELOC = system option specifies where the coverage analysis is to be written. The option takes either an external file specification enclosed in quotation marks or a SAS fileref.

MERROR System Option

Specifies whether the macro processor issues a warning message when a macro reference cannot be resolved.

Valid in:	Configuration file, OPTIONS window, OPTIONS statement, SAS invocation
Category:	Macro
PROC OPTIONS GROUP=	MACRO
Type:	System option

Default:	MERROR

Syntax

MERROR

NOMERROR

Syntax Description

MERROR
: issues the following warning message when the macro processor cannot match a macro reference to a compiled macro:

```
WARNING: Apparent invocation of macro %text not resolved.
```

NOMERROR
: issues no warning messages when the macro processor cannot match a macro reference to a compiled macro.

Details

Several conditions can prevent a macro reference from resolving. These conditions appear when

- a macro name is misspelled
- a macro is called before being defined
- strings containing percent signs are encountered. For example:

```
TITLE Cost Expressed as %Sales;
```

If your program contains a percent sign in a string that could be mistaken for a macro keyword, specify NOMERROR.

MEXECNOTE System Option

Specifies whether to display macro execution information in the SAS log at macro invocation.

Valid in:	Configuration file, OPTIONS window, OPTIONS statement, SAS invocation
Category:	Macro
PROC OPTIONS GROUP=	MACRO
Type:	System option
Default:	NOMEXECNOTE
See:	MEXECSIZE on page 451

Syntax

MEXECNOTE

NOMEXECNOTE

Syntax Description

MEXECNOTE
displays the macro execution information in the log when the macro is invoked.

NOMEXECNOTE
does not display the macro execution information in the log when the macro is invoked.

Details

The MEXECNOTE option controls the generation of a NOTE in the SAS log that indicates the macro execution mode.

MEXECSIZE= System Option

Specifies the maximum macro size that can be executed in memory.

Valid in:	Configuration file, OPTIONS window, OPTIONS statement, SAS invocation
Category:	Macro
PROC OPTIONS GROUP=	MACRO
Type:	System option
Default:	65536
See:	MEXECNOTE on page 450 and MCOMPILENOTE on page 444

Syntax

MEXECSIZE=*n* | *n*K | *n*M | *n*G | *n*T | *hex*X | MIN | MAX

Required Arguments

n
specifies the maximum size macro to be executed in memory available in bytes.

***n*K**
: specifies the maximum size macro to be executed in memory available in kilobytes.

***n*M**
: specifies the maximum size macro to be executed in memory available in megabytes.

***n*G**
: specifies the maximum size macro to be executed in memory available in gigabytes.

***n*T**
: specifies the maximum size macro to be executed in memory available in terabytes.

MIN
: specifies the minimum size macro to be executed in memory. Minimum value is 0.

MAX
: specifies the maximum size macro to be executed in memory. Maximum value is 2,147,483,647.

***hex*X**
: specifies the maximum size macro to be executed in memory by a hexadecimal number followed by an X.

Details

Use the MEXECSIZE option to control the maximum size macro that will be executed in memory as opposed to being executed from a file. The MEXECSIZE option value is the compiled size of the macro. Memory is allocated only when the macro is executed. After the macro completes, the memory is released. If memory is not available to execute the macro, an out-of-memory message is written to the SAS log. Use the MCOMPILENOTE option to write to the SAS log the size of the compiled macro. The MEMSIZE option does not affect the MEXECSIZE option.

MFILE System Option

Specifies whether MPRINT output is routed to an external file.

Valid in:	Configuration file, OPTIONS window, OPTIONS statement, SAS invocation
Category:	Macro
PROC OPTIONS GROUP=	MACRO
Type:	System option
Default:	NOMFILE

Requirement:	MPRINT option
Note:	MFILE is intended for debugging purposes only. Using it might negatively impact system performance.
See:	“MPRINT System Option” on page 463

Syntax

MFILE

NOMFILE

Syntax Description

MFILE
routes output produced by the MPRINT option to an external file. This option is useful for debugging.

NOMFILE
does not route MPRINT output to an external file.

Details

This option makes it easier to debug notes, warnings, and errors that are caused by non-macro code that is generated by a macro. When MFILE is in effect, all of the code that is written to the SAS log by the MPRINT option is also written to a file. You can then use the file to debug the generated code. To use MFILE, the MPRINT option must also be in effect, and an external file must be assigned the fileref MPrint. When these conditions are met, macro-generated code that is displayed by the MPRINT option in the SAS log during macro execution is written to the external file referenced by the fileref MPrint.

If MPrint is not assigned as a fileref or if the file cannot be accessed, warnings are written to the SAS log and MFILE is set to off. In that case, to use the feature, you must specify MFILE again and assign the fileref MPrint to a file that can be accessed.

Example: Debug DATA Step Code Generated by a Macro

This simple example shows you how to use the MFILE and MPRINT options to debug a problem in DATA step code that is generated by a macro. Here is the macro that generates the DATA step code:

```
%macro test;
   data a;
      x='This is a sample line of text';
```

```
        y=substr(x,99);
        put y;
        run;
%mend;

%test;
```

When this macro is invoked, the following note is written to the SAS log:

```
%test
NOTE: Invalid second argument to function SUBSTR at line 1 column 53.

x=This is a sample line of text y=  _ERROR_=1 _N_=1
```

To begin debugging this problem, use MFILE and MPRINT to write the DATA step code generated by macro %TEST to a file:

```
filename mprint 'output-file-name.sas';
options MPRINT MFILE;
%test;
options NOMPRINT NOMFILE;
```

When this code is executed, the following code is written to file *output-file-name*.sas:

```
data a;
x='This is a sample line of text';
y=substr(x,99);
put y;
run;
```

You can open file *output-file-name*.sas in SAS to begin debugging the generated code. In the generated code, the second argument in the SUBSTR function call is the starting position in the source string. The specified starting position 99 exceeds the length of the string that is currently stored in variable x, which makes 99 invalid. The starting position should be 9. To verify this fix, change the second argument in the SUBSTR function call in the generated code to 9, and then run the code. The modified DATA step code generates the correct result:

```
63   data a;
64   x='This is a sample line of text';
65   y=substr(x,9);
66   put y;
67   run;

a sample line of text
NOTE: The data set WORK.A has 1 observations and 2 variables.
```

Now that the problem has been identified and the fix has been verified, macro %TEST can be modified and tested:

```
%macro test;
   data a;
      x='This is a sample line of text';
      y=substr(x,9);
      put y;
      run;
%mend;
```

```
%test
```

It now generates the correct output.

```
a sample line of text
NOTE: The data set WORK.A has 1 observations and 2 variables.
```

Although this example is simple, the method used can be very helpful in debugging generated code in more complex cases.

MINDELIMITER= System Option

Specifies the character to be used as the delimiter for the macro IN operator.

Valid in:	Configuration file, OPTIONS window, OPTIONS statement, SAS invocation
Category:	Macro
PROC OPTIONS GROUP=	MACRO
Type:	System option
Default:	a blank
Restriction:	The following characters cannot be used as a delimiter `% & ' " ( ) ;`
Requirement:	In order to use this option, system option MINOPERATOR must be in effect. If system option NOMINOPERATOR is in effect, option MINDELIMITER= is ignored.
Note:	To specify a delimiter for a single macro definition, use the %MACRO statement MINDELIMITER= on page 407 option instead.
See:	"MINOPERATOR System Option" on page 457 and "%MACRO Macro Statement" on page 406

Syntax

MINDELIMITER="*single-character*"

Required Argument

single-character
is a single character enclosed in double or single quotation marks. The character will be used as the delimiter for the macro IN operator. Here is an example:

double quotation marks

```
mindelimiter=",";
```

or single quotation marks

```
mindelimiter=',';
```

Restriction The following characters cannot be used as a delimiter

```
% & ' " ( ) ;
```

Details

This option and the macro IN operator apply only when system option MINOPERATOR is in effect. The MINDELIMITER= option value is retained in original case (lowercase or uppercase) and can have a maximum length of one character. The default value of the MINDELIMITER option is a blank.

You can use the # character instead of IN.

Note: When the IN or # operator is used in a macro, the delimiter that is used at the execution time of the macro is the value of the MINDELIMITER option at the time of the compilation of the macro. A specific delimiter value for use during the execution of the macro other than the current value of the MINDELIMITER system option might be specified on the macro definition statement:

```
%macro macroname / mindelimiter=',';
```

Example

The following is an example using a specified delimiter in an IN operator:

```
option minoperator;

%put %eval(a in d,e,f,a,b,c); /* should print 0 */
%put %eval(a in d e f a b c); /* should print 1 */

option mindelimiter=',';

%put %eval(a in d,e,f,a,b,c); /* should print 1 */
%put %eval(a in d e f a b c); /* should print 0 */
```

The following is the output to the SAS log:

```
NOTE: Copyright (c) 2016 by SAS Institute Inc., Cary, NC, USA.
NOTE: SAS (r) Proprietary Software 9.4 (TS1M6)
      Licensed to SAS Institute Inc., Site 1.
NOTE: This session is executing on the X64_10PRO  platform.
NOTE: SAS initialization used:
      real time           19.22 seconds
      cpu time            2.26

option minoperator;

%put %eval(a in d,e,f,a,b,c); /* should print 0 */
0
  %put %eval(a in d e f a b c); /* should print 1 */
1
  option mindelimiter=',';

  %put %eval(a in d,e,f,a,b,c); /* should print 1 */
1
  %put %eval(a in d e f a b c); /* should print 0 */
0
```

See Also

"MINOPERATOR System Option" on page 457

MINOPERATOR System Option

Controls whether the macro processor recognizes and evaluates the IN (#) logical operator.

Valid in:	Configuration file, OPTIONS window, OPTIONS statement, SAS invocation
Category:	Macro
PROC OPTIONS GROUP=	MACRO
Type:	System option
Default:	NOMINOPERATOR
Note:	In order to use the macro IN operator in an expression, you must ensure that the MINOPERATOR system option has been changed from its default value of NOMINOPERATOR to MINOPERATOR. For more information, see MINOPERATOR system option on page 458 and "%MACRO Macro Statement" on page 406.

Syntax

MINOPERATOR

NOMINOPERATOR

Syntax Description

MINOPERATOR

causes the macro processor to recognize and evaluate both the mnemonic operator **`IN`** or the special character **`#`** as a logical operator in expressions.

NOMINOPERATOR

causes the macro processor to recognize both the mnemonic operator **`IN`** and the special character **`#`** as regular characters.

Details

Use the MINOPERATOR system option or in the %MACRO statement if you want to use the **`IN`** (**`#`**) as operators in expressions:

```
options minoperator;
```

To use IN or # as operators in expressions evaluated during the execution of a specific macro, use the MINOPERATOR keyword on the definition of the macro:

```
%macro macroname / minoperator;
```

The macro IN operator is similar to the DATA step IN operator, but not identical. The following is a list of differences:

- The macro IN operator cannot search a numeric array.
- The macro IN operator cannot search a character array.
- A colon (**`:`**) is not recognized as a shorthand notation to specify a range, such as **`1:10`** means 1 through 10. Instead, you use the following in a macro:

  ```
  %eval(3 in 1 2 3 4 5 6 7 8 9 10);
  ```

- The default delimiter for list elements is a blank. For more information, see "MINDELIMITER= System Option" on page 455.
- Both operands must contain a value.

  ```
  %put %eval(a IN a b c d); /*Both operands are present. */
  ```

  If an operand contains a null value, an error is generated.

  ```
  %put %eval(  IN a b c d); /*Missing first operand. */
  ```

  or

  ```
  %put %eval(a IN); /*Missing second operand. */
  ```

  Whether the first or second operand contains a null value, the same error is written to the SAS log:

  ```
  ERROR: Operand missing for IN operator in argument to %EVAL function.
  ```

The following example uses the macro IN operator to search a character string:

```
%if &state in (NY NJ PA) %then %let region = %eval(&region + 1);
```

For more information, see "Defining Arithmetic and Logical Expressions" on page 86.

See Also

"MINDELIMITER= System Option" on page 455

MLOGIC System Option

Specifies whether the macro processor traces its execution for debugging.

Valid in:	Configuration file, OPTIONS window, OPTIONS statement, SAS invocation
Category:	Macro
PROC OPTIONS GROUP=	MACRO LOGCONTROL
Type:	System option
Default:	NOMLOGIC

Syntax

MLOGIC

NOMLOGIC

Syntax Description

MLOGIC
causes the macro processor to trace its execution and to write the trace information to the SAS log. This option is a useful debugging tool.

Note You must have a good understanding of the macro source code to use this option.

NOMLOGIC
does not trace execution. Use this option unless you are debugging macros.

Details

Use MLOGIC to debug macros. Each line generated by the MLOGIC option is identified with the prefix MLOGIC(*macro-name*):. If MLOGIC is in effect and the macro processor encounters a macro invocation, the macro processor displays messages that identify the following:

- the beginning of macro execution
- values of macro parameters at invocation
- execution of each macro program statement

- whether each %IF condition is true or false
- the ending of macro execution

Note: Using MLOGIC can produce a great deal of output.

For more information about macro debugging, see Chapter 10, “Macro Facility Error Messages and Debugging,” on page 145 .

Example: Tracing Macro Execution

In this example, MLOGIC traces the execution of the macros MKTITLE and RUNPLOT:

```
%macro mktitle(proc,plotType,data);
    title "%upcase(&proc) &plotType of %upcase(&data)";
%mend mktitle;

%macro runplot(ds, type=bar);
   %let type = %upcase(&type);
   %if &type eq BAR %then %do;
      %mktitle (sgplot,Bar Chart,&ds)
      proc sgplot data=&ds;
         vbar style / response=price;
         label style="Style" price="Price";
      run;
   %end;
   %else %if &type eq SCATTER %then %do;
      %mktitle (sgplot,Scatter Plot,&ds)
      proc sgplot data=&ds;
         scatter y=style x=price;
         label style="Style" price="Price";
      run;
   %end;
   %else %put ERROR: Invalid plot type &type..;
%mend runplot;

options mlogic;
%runplot(Sasuser.Houses, type=scatter)
options nomlogic;
```

See Example Code 10.1 on page 151. When this program executes, this MLOGIC output is written to the SAS log:

```
MLOGIC(RUNPLOT):  Beginning execution.
MLOGIC(RUNPLOT):  Parameter DS has value Sasuser.Houses
MLOGIC(RUNPLOT):  Parameter TYPE has value scatter
MLOGIC(RUNPLOT):  %LET (variable name is TYPE)
MLOGIC(RUNPLOT):  %IF condition &type eq BAR is FALSE
MLOGIC(RUNPLOT):  %IF condition &type eq SCATTER is TRUE
MLOGIC(MKTITLE):  Beginning execution.
MLOGIC(MKTITLE):  Parameter PROC has value sgplot
MLOGIC(MKTITLE):  Parameter PLOTTYPE has value Scatter Plot
MLOGIC(MKTITLE):  Parameter DATA has value Sasuser.Houses
MLOGIC(MKTITLE):  Ending execution.
```

MLOGICNEST System Option

Specifies whether to display the macro nesting information in the MLOGIC output in the SAS log.

Valid in:	Configuration file, OPTIONS window, OPTIONS statement, SAS invocation
Category:	Macro
PROC OPTIONS GROUP=	MACRO LOGCONTROL
Type:	System option
Default:	NOMLOGICNEST

Syntax

MLOGICNEST

NOMLOGICNEST

Syntax Description

MLOGICNEST
enables the macro nesting information to be displayed in the MLOGIC output in the SAS log.

NOMLOGICNEST
prevents the macro nesting information from being displayed in the MLOGIC output in the SAS log.

Details

MLOGICNEST enables the macro nesting information to be written to the SAS log in the MLOGIC output.

The setting of MLOGICNEST does not affect the output of any currently executing macro.

The setting of MLOGICNEST does not imply the setting of MLOGIC. You must set both MLOGIC and MLOGICNEST in order for output (with nesting information) to be written to the SAS log.

Example: Using MLOGICNEST System Option

The first example shows both the MLOGIC and MLOGICNEST options being set:

```
%macro outer;
    %put THIS IS OUTER;
    %inner;
%mend outer;
%macro inner;
    %put THIS IS INNER;
    %inrmost;
%mend inner;
%macro inrmost;
    %put THIS IS INRMOST;
%mend;

options mlogic mlogicnest;
%outer
```

Here is the MLOGIC output in the SAS log using the MLOGICNEST option:

```
MLOGIC(OUTER):  Beginning execution.
MLOGIC(OUTER):  %PUT THIS IS OUTER
THIS IS OUTER
MLOGIC(OUTER.INNER):  Beginning execution.
MLOGIC(OUTER.INNER): %PUT THIS IS INNER
THIS IS INNER
MLOGIC(OUTER.INNER.INRMOST):  Beginning execution.
MLOGIC(OUTER.INNER.INRMOST):  %PUT THIS IS INRMOST
THIS IS INRMOST
MLOGIC(OUTER.INNER.INRMOST): Ending execution.
MLOGIC(OUTER.INNER):  Ending execution.
MLOGIC(OUTER):  Ending execution.
```

The second example uses only the NOMLOGICNEST option:

```
 %macro outer;
     %put THIS IS OUTER;
     %inner;
 %mend outer;
 %macro inner;
     %put THIS IS INNER;
     %inrmost;
 %mend inner;
 %macro inrmost;
     %put THIS IS INRMOST;
 %mend;

options nomlogicnest;
%outer
```

Here is the output in the SAS log when you use only the NOMLOGICNEST option:

```
MLOGIC(OUTER):  Beginning execution.
MLOGIC(OUTER):  %PUT THIS IS OUTER
THIS IS OUTER
MLOGIC(INNER):  Beginning execution.
MLOGIC(INNER):  %PUT THIS IS INNER
THIS IS INNER
MLOGIC(INRMOST):  Beginning execution.
MLOGIC(INRMOST):  %PUT THIS IS INRMOST
THIS IS INRMOST
MLOGIC(INRMOST):  Ending execution.
MLOGIC(INNER):  Ending execution.
MLOGIC(OUTER):  Ending execution.
```

MPRINT System Option

Specifies whether SAS statements that are generated by macro execution are written to the SAS log.

Valid in:	Configuration file, OPTIONS window, OPTIONS statement, SAS invocation
Category:	Macro
PROC OPTIONS GROUP=	MACRO LOGCONTROL
Type:	System option
Default:	NOMPRINT
See:	"MFILE System Option" on page 452

Syntax

MPRINT

NOMPRINT

Syntax Description

MPRINT
displays the SAS statements that are generated by macro execution. The SAS statements are useful for debugging macros.

Tip If you are not familiar with a macro's source code, use MPRINT to determine which SAS statements it generates during execution.

See "Example 1: Tracing Generation of SAS Statements" on page 464

"Example 2: Directing MPRINT Output to an External File" on page 465

"Examining the Generated SAS Statements with MPRINT" on page 165

MLOGIC on page 459 and SYMBOLGEN on page 477 for more advanced debugging options.

NOMPRINT
does not display SAS statements that are generated by macro execution.

Details

The MPRINT option displays the SAS statements that are generated by macro execution. This display enables you to determine the SAS statements that a macro generates without having any knowledge of the macro source code. It is useful in debugging SAS code that is generated by your macros. Each SAS statement in the MPRINT display begins a new line. Each line of MPRINT output is identified with the prefix MPRINT(*macro-name*):, to identify the macro that generates the statement. Tokens that are separated by multiple spaces are printed with one intervening space.

You can direct MPRINT output to an external file by also using the MFILE option and assigning the fileref MPrint to that file. For more information, see "MFILE System Option" on page 452.

Examples

Example 1: Tracing Generation of SAS Statements

In this example, MPRINT traces the SAS statements that are generated when the macros MKTITLE and RUNPLOT execute:

```
%macro mktitle(proc,plotType,data);
    title "%upcase(&proc) &plotType of %upcase(&data)";
%mend mktitle;

%macro runplot(ds, type=bar);
   %let type = %upcase(&type);
   %if &type eq BAR %then %do;
      %mktitle (sgplot,Bar Chart,&ds)
      proc sgplot data=&ds;
         vbar style / response=price;
         label style="Style" price="Price";
      run;
   %end;
   %else %if &type eq SCATTER %then %do;
      %mktitle (sgplot,Scatter Plot,&ds)
      proc sgplot data=&ds;
         scatter y=style x=price;
         label style="Style" price="Price";
      run;
   %end;
   %else %put ERROR: Invalid plot type &type..;
%mend runplot;

options mprint;
%runplot(Sasuser.Houses, type=scatter)
options nomprint;
```

See Example Code 10.1 on page 151. When this program executes, this MPRINT output is written to the SAS log:

```
MPRINT(MKTITLE):   title "SGPLOT Scatter Plot of SASUSER.HOUSES";
MPRINT(RUNPLOT):   proc sgplot data=Sasuser.Houses;
MPRINT(RUNPLOT):   scatter y=style x=price;
MPRINT(RUNPLOT):   label style= "Style" price= "Price";
MPRINT(RUNPLOT):   run;
```

Example 2: Directing MPRINT Output to an External File

Adding these statements before the macro call in the previous program sends the MPRINT output to the file DebugMac when the Compute Server session ends.

```
options mfile mprint;
filename mprint 'debugmac';
```

For another example of using MPRINT and MFILE, see “Example: Debug DATA Step Code Generated by a Macro” on page 453.

MPRINTNEST System Option

Specifies whether to display the macro nesting information in the MPRINT output in the SAS log.

Valid in:	Configuration file, OPTIONS window, OPTIONS statement, SAS invocation
Category:	Macro
PROC OPTIONS GROUP=	MACRO
Type:	System option
Default:	NOMPRINTNEST

Syntax

MPRINTNEST

NOMPRINTNEST

Syntax Description

MPRINTNEST
enables the macro nesting information to be displayed in the MPRINT output in the SAS log.

NOMPRINTNEST
prevents the macro nesting information from being displayed in the MPRINT output in the SAS log.

Details

MPRINTNEST enables the macro nesting information to be written to the SAS log in the MPRINT output. The MPRINTNEST output has no effect on the MPRINT output that is sent to an external file. For more information, see MFILE System Option.

The setting of MPRINTNEST does not imply the setting of MPRINT. You must set both MPRINT and MPRINTNEST in order for output (with the nesting information) to be written to the SAS log.

Example: Using MPRINTNEST System Option

The following example uses the MPRINT and MPRINTNEST options:

```
%macro outer;
data _null_;
     %inner
run;
%mend outer;
%macro inner;
    put %inrmost;
%mend inner;
%macro inrmost;
    'This is the text of the PUT statement'
%mend inrmost;

options mprint mprintnest;
%outer
```

Here is the output written to the SAS log using both the MPRINT option and the MPRINTNEST option:

```
MPRINT(OUTER):   data _null_;
MPRINT(OUTER.INNER):   put
MPRINT(OUTER.INNER.INRMOST):   'This is the text of the PUT statement'
MPRINT(OUTER.INNER):   ;
MPRINT(OUTER):   run;
This is the text of the PUT statement
NOTE: DATA statement used (Total process time):
      real time           0.10 seconds
      cpu time            0.06 seconds
```

Here is an example that uses the NOMPRINTNEST option:

```
%macro outer;
    data _null_;
    %inner
run;
%mend outer;
%macro inner;
    put %inrmost;
%mend inner;
%macro inrmost;
```

```
    'This is the text of the PUT statement'
%mend inrmost;

options nomprintnest;
%outer
```

Here is the output written to the SAS log using the NOMPRINTNEST option:

```
MPRINT(OUTER):   data _null_;
MPRINT(INNER):   put
MPRINT(INRMOST):   'This is the text of the PUT statement'
MPRINT(INNER):  ;
MPRINT(OUTER):   run;
This is the text of the PUT statement
NOTE: DATA statement used (Total process time):
      real time           0.00 seconds
      cpu time            0.01 seconds
```

MRECALL System Option

Specifies whether autocall libraries are searched for a member that was not found during an earlier search.

Valid in:	Configuration file, OPTIONS window, OPTIONS statement, SAS invocation
Category:	Macro
PROC OPTIONS GROUP=	MACRO
Type:	System option
Default:	NOMRECALL

Syntax

MRECALL

NOMRECALL

Syntax Description

MRECALL
searches the autocall libraries for an undefined macro name each time an attempt is made to invoke the macro. It is inefficient to search the autocall libraries repeatedly for an undefined macro. Generally, use this option when you are developing or debugging programs that call autocall macros.

NOMRECALL
searches the autocall libraries only once for a requested macro name.

Details

Use the MRECALL option primarily for

- developing systems that require macros in autocall libraries.
- recovering from errors caused by an autocall to a macro that is in an unavailable library. Use MRECALL to call the macro again after making the library available. In general, do not use MRECALL unless you are developing or debugging autocall macros.

MREPLACE System Option

Specifies whether to enable existing macros to be redefined.

Valid in:	Configuration file, OPTIONS window, OPTIONS statement, SAS invocation
Category:	Macro
PROC OPTIONS GROUP=	MACRO
Type:	System option
Default:	MREPLACE

Syntax

MREPLACE

NOMREPLACE

Syntax Description

MREPLACE
enables you to redefine existing macro definitions that are stored in a catalog in the Work library.

NOMREPLACE
prevents you from redefining existing macro definitions that are stored in a catalog in the Work library.

Details

The MREPLACE system option enables you to overwrite existing macros if the names are the same.

The NOMREPLACE system option prevents you from overwriting a macro even if a macro with the same name has already been compiled.

MSTORED System Option

Specifies whether the macro facility searches a specific catalog for a stored compiled macro.

Valid in:	Configuration file, OPTIONS window, OPTIONS statement, SAS invocation
Category:	Macro
PROC OPTIONS GROUP=	MACRO
Type:	System option
Default:	NOMSTORED

Syntax

MSTORED

NOMSTORED

Syntax Description

MSTORED
searches for stored compiled macros in a catalog in the SAS library referenced by the SASMSTORE= option.

NOMSTORED
does not search for compiled macros.

Details

Regardless of the setting of MSTORED, the macro facility first searches for macros compiled in the current Compute Server session. If the MSTORED option is in effect, the macro facility next searches the libraries containing compiled stored macros. If the MAUTOSOURCE option is in effect, the macro facility next searches the autocall libraries. Then, the macro facility searches the SASMacr catalog in the SASHelp library.

MSYMTABMAX= System Option

Specifies the maximum amount of memory available to all macro variable symbol tables (global and local, combined).

Valid in:	SAS Environment Manager, OPTIONS statement, SASV9_OPTIONS environment variable
Category:	Macro
PROC OPTIONS GROUP=	MACRO
Type:	System option
Default:	4M
Linux specifics:	default value
Note:	This option cannot be restricted by a site administrator. For more information, see "Using SAS System Options" in *SAS System Options: Reference*.

Syntax

MSYMTABMAX= *n* | *n*K | *n*M | *n*G | *hex*X | MIN | MAX

Required Arguments

***n* | *n*K | *n*M | *n*G**

specifies the maximum amount of memory that is available in multiples of 1 (bytes); 1,024 (kilobytes); 1,048,576 (megabytes); or 1,073,741,824 (gigabytes). You can specify decimal values for the number of kilobytes, megabytes, or gigabytes. For example, a value of 8 specifies 8 bytes, a value of .782k specifies 801 bytes, and a value of 3m specifies 3,145,728 bytes.

Note: You can also specify the KB, MB, or GB syntax notations.

***hex*X**

specifies the maximum amount of memory that is available as a hexadecimal value. You must specify the value beginning with a number (0–9), followed by hexadecimal characters (0–9, A–F), and then followed by an X. For example, 2dx sets the maximum amount of memory to 45 bytes.

MIN

sets the amount of memory that is available to the minimum setting, which is 0 bytes. Setting the amount of memory to the minimum setting causes all macro symbol tables to be written to disk.

MAX

sets the amount of memory that is available to the maximum setting. On 64-bit computers, this value is 9,007,199,254,740,992 bytes.

Details

Once the maximum value is reached, additional macro variables are written out to disk.

The value that you specify with the MSYMTABMAX= system option can range from 0 to the largest nonnegative integer representable on your operating environment. The default values are host dependent. A value of 0 causes all macro symbol tables to be written to disk.

The value of MSYMTABMAX= can affect system performance. If this option is set too low and the application frequently reaches the specified memory limit, then disk I/O increases. If this option is set too high (on some operating environments) and the application frequently reaches the specified memory limit, then less memory is available for the application, and CPU usage increases. Before you specify the value for production jobs, run tests to determine the optimum value.

MVARSIZE= System Option

Specifies the maximum number of bytes for any macro variable stored in memory.

Valid in:	SAS Environment Manager, OPTIONS statement, SASV9_OPTIONS environment variable
Category:	Macro
PROC OPTIONS GROUP=	MACRO
Type:	System option
Default:	65534
Linux specifics:	default value
Note:	This option cannot be restricted by a site administrator. For more information, see "Using SAS System Options" in *SAS System Options: Reference*.

Syntax

MVARSIZE=*n* | *n*K | *n*M | *n*G | *hex*X | MIN | MAX

Required Arguments

***n* | *n*K | *n*M | *n*G**

specifies the maximum macro variable size in multiples of 1 (bytes); 1,024 (kilobytes); 1,048,576 (megabytes); or 1,073,741,824 (gigabytes). You can specify decimal values for the number of kilobytes, megabytes, or gigabytes. For example, a value of 8 specifies 8 bytes, a value of .782k specifies 801 bytes, and a value of 3m specifies 3,145,728 bytes.

Note: You can also specify the KB, MB, or GB syntax notations.

***hex*X**
: specifies the maximum macro variable size as a hexadecimal value. You must specify the value beginning with a number (0–9), followed by hexadecimal characters (0–9, A–F), and then followed by an X. For example, 2dx sets the maximum macro variable size to 45 bytes.

MIN
: sets the macro variable size to the minimum setting, which is 0 bytes. Setting the macro variable size to the minimum setting causes all macro variable values to be written to disk.

MAX
: sets the macro variable size to the maximum setting, which is 65,534 bytes.

Details

If the memory required for a macro variable value is larger than the MVARSIZE= value, the variable is written to a temporary catalog on disk. The macro variable name is used as the member name, and all members have the type MSYMTAB.

The value that you specify with the MVARSIZE= system option can range from 0 to 65534. A value of 0 causes all macro variable values to be written to disk.

The value of MVARSIZE= can affect system performance. If this option is set too low and the application frequently creates macro variables larger than the limit, then disk I/O increases. Before you specify the value for production jobs, run tests to determine the optimum value.

Note: The MVARSIZE= option has no affect on the maximum length of the value of the macro variable. For more information, see "Macro Variables" on page 27.

SASAUTOS= System Option

Specifies the location of one or more autocall libraries.

Valid in:	SAS Environment Manager, OPTIONS statement, SASV9_OPTIONS environment variable
Categories:	Environment Control: Files Macro
PROC OPTIONS GROUP=	ENVFILES MACRO
Type:	System option
Default:	SASAUTOS fileref
Linux specifics:	syntax for specifying multiple *directory-specifications*

Note: This option can be restricted by a site administrator. For more information, see "Using SAS System Options" in *SAS System Options: Reference*.

Syntax

SASAUTOS='*directory-specification*' | *fileref*

SASAUTOS=('*directory-specification-1*' | *fileref-1*, ..., '*directory-specification-n*' | *fileref-n*)

NOSASAUTOS

Required Arguments

directory-specification
specifies a pathname to an autocall macro library.

fileref
specifies a name (shorthand reference) that has been assigned to an autocall macro library.

Note that the SASAUTOS option uses filerefs, not librefs.

Details

About Autocall Libraries

An autocall library contains files that define SAS macros. The following sections discuss aspects of autocall libraries that are dependent on the operating environment.

Available Autocall Macros

There are two types of autocall macros, those macros that are provided by SAS, and those macros that you define yourself. To use the autocall facility, you must have the MAUTOSOURCE system option set.

When SAS is installed, the SASAUTOS system option is defined in the configuration file to refer to the location of the default macros that are supplied by SAS. The products licensed at your site determine the autocall macros that you have available. You can also define your own autocall macros and store them in one or more directories. SAS does not recognize autocall macros if their filenames are in uppercase or in mixed case. Use only lowercase for filenames.

Guidelines for Naming Macro Files

Macro names in SAS are case insensitive, but they all map to a lowercase filename. If you store autocall macros in a Linux directory, the file extension must be `.sas`, and the filename must be entirely in lowercase. In the Linux environment, each macro file in the directory must contain a macro definition with a macro name that matches the filename. For example, a file named **`prtdata.sas`** should define a macro named **`prtdata`**.

Using the SASAUTOS System Option

To use your own autocall macros in your SAS program, specify their directories with the SASAUTOS system option.

Note: The SASAUTOS system option under Linux does not recognize filenames that are in uppercase or mixed case.

You can set the SASAUTOS system option in an OPTIONS statement during your SAS session. However, autocall libraries that are specified with the OPTIONS statement override any previous specification, such as values set in SAS Environment Manager. Each autocall macro library consists of files in a Linux directory. The *directory-specification* can be the pathname of a Linux directory, a fileref, or an environment variable.

If you specify the pathname of a directory, you must enclose the name in quotation marks. You can omit the quotation marks only if you are specifying the name in the configuration file via SAS Environment Manager or in the SASV9_OPTIONS environment variable. And, you can omit the quotation marks only if the name cannot be interpreted as a fileref.

If you specify a fileref, you must define it before attempting to use any of the autocall macros. You can define the fileref in a FILENAME statement, in an environment variable, or with the FILENAME function. See *SAS Programmer's Guide: Essentials*.

How you specify multiple directory names, filerefs, or environment variables depends on where you specify the SASAUTOS option:

- If you specify the SASAUTOS option in the configuration file or in the SASV9_OPTIONS environment variable, use either multiple SASAUTOS options, or enclose the directory names in parentheses. Separate the names with a comma or a blank space.
- If you specify the SASAUTOS option in the OPTIONS statement or in the SAS System Options window, you must enclose the directory names in parentheses. Separate the names with a comma or a blank space.

At configuration time, SAS concatenates all directories specified for SASAUTOS. However, after the session starts, any new directories that you specify override any current autocall libraries.

The NOSASAUTOS option causes SAS to ignore all previous SASAUTOS specifications (whether specified in the configuration file or in the SASV9_OPTIONS environment variable).

The default value of the SASAUTOS option is the SASAUTOS fileref. There is no Linux directory assigned to the fileref, so you must define the SASAUTOS fileref if you want to use it as your autocall library.

Examples

Example 1: Specifying Multiple Environment Variables in the OPTIONS Statement

The following example shows the syntax to use if you are specifying multiple environment variables in the OPTIONS statement:

```
options sasautos=(AUTODIR, SASAUTOS);
```

The environment variables that you specify must be defined. For example, you could define the AUTODIR environment variable as a SAS command by using the following code:

```
set=AUTODIR /tmp/sasautos
```

For more information about how to define an environment variable, see "SET System Option" in *SAS System Options: Reference*.

Example 2: Specifying a Fileref in the OPTIONS Statement

The fileref that you specify must be defined. For example, you could define the AUTODIR fileref using a FILENAME statement:

```
filename AUTODIR '/tmp/sasautos';
```

Once the fileref is defined, you can use it in an OPTIONS statement to set the autocall library.

```
options sasautos=autodir;
```

See Also

System Options:

- "APPEND= System Option" in *SAS System Options: Reference*
- "INSERT= System Option" in *SAS System Options: Reference*
- "MAUTOSOURCE System Option" on page 442
- "MRECALL System Option" on page 467

SASMSTORE= System Option

Identifies the libref of a SAS library with a catalog that contains, or will contain, stored compiled SAS macros.

Valid in:	Configuration file, OPTIONS window, OPTIONS statement, SAS invocation
Category:	Macro

PROC OPTIONS GROUP=	MACRO
Type:	System option

Syntax

SASMSTORE=*libref*

Required Argument

libref
specifies the libref of a SAS library that contains, or will contain, a catalog of stored compiled SAS macros. This libref cannot be Work.

SERROR System Option

Specifies that the macro processor issues a warning message when a macro variable reference does not match a macro variable.

Valid in:	Configuration file, OPTIONS window, OPTIONS statement, SAS invocation
Category:	Macro
PROC OPTIONS GROUP=	MACRO
Type:	System option
Default:	SERROR

Syntax

SERROR

NOSERROR

Syntax Description

SERROR
issues a warning message when the macro processor cannot match a macro variable reference to an existing macro variable.

Alias SERR

NOSERROR
issues no warning messages when the macro processor cannot match a macro variable reference to an existing macro variable.

Alias NOSERR

Details

Several conditions can occur that prevent a macro variable reference from resolving. These conditions appear when one or more of the following is true:

- the name in a macro variable reference is misspelled.
- the variable is referenced before being defined.
- the program contains an ampersand (&) followed by a string, without intervening blanks between the ampersand and the string. For example:

```
if x&y then do;
if buyer="Smith&Jones, Inc." then do;
```

If your program uses a text string containing ampersands and you want to suppress the warnings, specify NOSERROR.

SYMBOLGEN System Option

Specifies that the results of resolving macro variable references are written to the SAS log for debugging.

Valid in:	Configuration file, OPTIONS window, OPTIONS statement, SAS invocation
Category:	Macro
PROC OPTIONS GROUP=	MACRO LOGCONTROL
Type:	System option
Default:	NOSYMBOLGEN

Syntax

SYMBOLGEN

NOSYMBOLGEN

Syntax Description

SYMBOLGEN
displays the results of resolving macro variable references. This option is useful for debugging.

Alias SGEN

Note You must have a good understanding of the macro source code to use this option.

NOSYMBOLGEN
does not display results of resolving macro variable references.

Alias NOSGEN

Details

SYMBOLGEN displays the results in this form:

```
SYMBOLGEN: Macro variable name resolves to value
```

SYMBOLGEN also indicates when a double ampersand (&&) resolves to a single ampersand (&).

Example: Tracing Resolution of Macro Variable References

In this example, SYMBOLGEN traces the resolution of macro variable references when the macros MKTITLE and RUNPLOT execute:

```
%macro mktitle(proc,plotType,data);
    title "%upcase(&proc) &plotType of %upcase(&data)";
%mend mktitle;

%macro runplot(ds, type=bar);
   %let type = %upcase(&type);
   %if &type eq BAR %then %do;
      %mktitle (sgplot,Bar Chart,&ds)
      proc sgplot data=&ds;
         vbar style / response=price;
         label style="Style" price="Price";
      run;
   %end;
   %else %if &type eq SCATTER %then %do;
      %mktitle (sgplot,Scatter Plot,&ds)
      proc sgplot data=&ds;
         scatter y=style x=price;
         label style="Style" price="Price";
      run;
   %end;
   %else %put ERROR: Invalid plot type &type..;
%mend runplot;

options symbolgen;
%runplot(Sasuser.Houses, type=scatter)
options nosymbolgen;
v
```

See Example Code 10.1 on page 151. When this program executes, this SYMBOLGEN output is written to the SAS log:

```
SYMBOLGEN:  Macro variable TYPE resolves to scatter
SYMBOLGEN:  Macro variable TYPE resolves to SCATTER
SYMBOLGEN:  Macro variable TYPE resolves to SCATTER
SYMBOLGEN:  Macro variable DS resolves to Sasuser.Houses
SYMBOLGEN:  Macro variable PROC resolves to sgplot
SYMBOLGEN:  Macro variable PLOTTYPE resolves to Scatter Plot
SYMBOLGEN:  Macro variable DATA resolves to Sasuser.Houses
SYMBOLGEN:  Macro variable DS resolves to Sasuser.Houses
```

SYSPARM= System Option

Specifies a character string that can be passed to SAS programs.

Valid in:	Configuration file, OPTIONS window, OPTIONS statement, SAS invocation
Category:	Macro
Type:	System option

Syntax

SYSPARM='*character-string*'

Required Argument

character-string
is a character string, enclosed in quotation marks, with a maximum length of 32767.

Details

The character string specified can be accessed in a SAS DATA step by the SYSPARM() function or anywhere in a SAS program by using the automatic macro variable reference &SYSPARM.

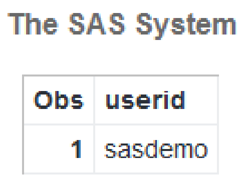
Operating Environment Information: The syntax shown here applies to the OPTIONS statement. At invocation, on the command line, or in a configuration file, the syntax is host specific. For more information, see the SAS documentation for your operating environment.

Example: Passing a User Identification to a Program

This example uses the SYSPARM option to pass a user identification to a program.

```
options sysparm='sasdemo';
data a;
   length userid $100;
   if sysparm()='sasdemo' then userid="&sysparm";
run;

proc print data=a;
run;
```

Output 20.1 *PRINT Procedure Output*



The SAS System

Obs	userid
1	sasdemo

PART 3

Appendixes

Appendix 1
Reserved Words in the Macro Facility

Macro Facility Word Rules

The following rules apply to the macro facility.

- Do not use a reserved word as the name of a macro, a macro variable, or a macro label. Reserved words include words reserved by both the macro facility and the operating environment. When a macro name is a macro facility reserved word, the macro processor issues a warning, and the macro is neither compiled nor available for execution. The macro facility reserves the words listed under "Reserved Words" on page 483 for internal use.
- Do not prefix the name of a macro language element with SYS because SAS reserves the SYS prefix for the names of macro language elements supplied with SAS software.
- Do not prefix macro variables names with SYS, AF, or FSP in order to avoid macro name conflicts.

Reserved Words

The following is a list of reserved words for the macro facility.

ABEND	GO	NRBQUOTE	STR
ABORT	GOTO	NRQUOTE	SUBSTR
ACT	IF	NRSTR	SUPERQ
ACTIVATE	INC	ON	SYMDEL
BQUOTE	INCLUDE	OPEN	SYMGLOBL
BY	INDEX	PAUSE	SYMLOCAL
CLEAR	INFILE	PUT	SYMEXIST
CLOSE	INPUT	QKCMPRES	SYSCALL
CMS	KCMPRES	QKLEFT	SYSEVALF
COMANDR	KINDEX	QKSCAN	SYSEXEC
COPY	KLEFT	QKSUBSTR	SYSFUNC
DEACT	KLENGTH	QKTRIM	SYSGET
DEL	KSCAN	QKUPCASE	SYSRPUT
DELETE	KSUBSTR	QSCAN	THEN
DISPLAY	KTRIM	QSUBSTR	TO
DMIDSPLY	KUPCASE	QSYSFUNC	TSO
DMISPLIT	LENGTH	QUOTE	UNQUOTE
DO	LET	QUPCASE	UNSTR
EDIT	LIST	RESOLVE	UNTIL
ELSE	LISTM	RETURN	UPCASE
END	LOCAL	RUN	WHILE
EVAL	MACRO	SAVE	WINDOW
FILE	MEND	SCAN	
GLOBAL	METASYM	STOP	

Appendix 2
SAS Macro Facility Error and Warning Messages

SAS Macro Error Messages

This section contains error messages that might be reported when using macros and the solutions to correct them. If you are unable to resolve the error, then contact SAS Technical Support.

`ERROR: %EVAL must be followed by an expression enclosed in parentheses.`

Cause	Solution
There is no expression following the %EVAL function.	The %EVAL function must be followed by an expression that is enclosed in parentheses .
An expression follows the %EVAL function but the parentheses are missing.	The %EVAL function must contain an expression that is enclosed in parentheses.

`ERROR: Expecting a variable name after value.`

Cause	Solution
The first letter of the variable name begins with a number or special character.	The first character must begin with an English letter or an underscore. Subsequent characters can be English letters, digits, or underscores.
The variable name is missing.	Verify that the variable name exists.

```
ERROR: Symbolic variable name value must be 32 or fewer characters
long.
```

Cause	Solution
The macro variable name is longer than 32 characters.	A macro variable name can be only 32 characters or less.

```
ERROR: Symbolic variable name value must begin with a letter or
underscore.
```

Cause	Solution
The macro variable name contained within quotation marks in the first argument to the SYMPUT routine or the SYMPUTX routine begins with a number or special character.	The macro variable name must begin with a letter or an underscore.
The value of the SAS data set variable that is used to create the macro variable name begins with a number or special character.	The value within the data set variable must begin with a letter or underscore.

```
ERROR: Symbolic variable name value must contain only letters,
digits, and underscores.
```

Cause	Solution
The macro variable name contained within quotation marks in the first argument to the SYMPUT routine or the SYMPUTX routine contains a special character.	The macro variable name must contain only letters, digits, or underscores.
The value of the SAS data set variable that is used to create the macro variable name within the SYMPUT routine or the SYMPUTX routine contains a special character.	The value within the data set variable must contain only letters, digits, or underscores.

`ERROR: Expected open parenthesis after macro function name not found.`

Cause	Solution
The macro function being used is missing the open parenthesis to begin the argument.	Each macro function used must contain an open and close parenthesis.

`ERROR: Maximum level of nesting of macro functions exceeded.`

Cause	Solution
The macro functions are nested more than ten times.	You cannot nest macro functions more than ten times.
An attempt was made to assign a macro variable to itself. A macro quoting function was used inside a %DO loop that is going beyond ten iterations.	Many times a macro quoting function is used when it is not needed. If this is the case, then remove the quoting function.

`ERROR: Invalid macro name value. It should be a valid SAS identifier no longer than 32 characters.`

Cause	Solution
The macro name begins with a character other than an underscore or letter.	Remove the special character at the beginning of the macro name so that it begins with an underscore or letter.
There is a period within the macro name.	Remove the period contained within the macro name.
The macro name is longer than 32 characters.	Reduce the length of the macro name to 32 characters or less.

`ERROR: A character operand was found in the %EVAL function or %IF condition where a numeric operand is required. The condition was: value`

Cause	Solution
A %EVAL function or a %IF statement contains an expression that has a negative floating-point number, for example, `%eval(-1.2 le 2)`	A %IF statement uses an implied %EVAL function that can handle only whole numbers. Use the %SYSEVALF function around the expression instead, for example, `%sysevalf(-1.2 le 2)`

Cause	Solution
The start or stop value contained within a %DO loop contains a character other than an integer.	The start and stop values within a %DO loop must be integers or macro expressions that generate integers.
There is a non-integer value within an arithmetic expression, for example, `%eval(3.2+2)`	The %EVAL function evaluates arithmetic and logical expressions. If arithmetic is taking place, then all values must be whole numbers. If you are using floating-point values, then use %SYSEVALF, for example, `%sysevalf(3.2+2)`
There is a letter or special character within the %EVAL function where an integer was expected, for example, `%if a+2 = 4 %then %put test;`	Remove the non-integer value. If you are trying to reference a SAS data set variable, then change the code to a DATA step IF condition.
A non-existent macro variable is referenced on a %IF or %DO statement.	Make sure the macro variable that is referenced exists and contains a valid value for the expression.

`ERROR: More positional parameters found than defined.`

Cause	Solution
The macro was defined with **n** number of parameters but the invocation went beyond that number.	If the definition contains two positional parameters, then the invocation must contain two positional parameters.
A macro variable on the invocation contains a comma within the value.	The macro variable that contains the comma as text must be surrounded with a macro quoting function, such as the %BQUOTE function. Here is an example: `%test(%bquote(&var),b,c)`. The example code contains 3 parameters values: **&var**, **b**, and **c**.
The string being passed as a parameter contains a comma.	The string that contains the comma needs a quoting function to hide the comma, such as the %STR function. Here is an example: `%test(%str(a,b),c)`. The example code contains three parameters: **a,b** and **c**.
The %STR function was used to mask the comma.	The %BQUOTE function or the %SUPERQ function should be used instead.

`ERROR: Expecting comma (to separate macro parameters) or close parenthesis (to end parameter list) but found: value`

Cause	Solution
A positional parameter precedes a keyword parameter. There is a missing comma between the two parameters, for example, `%macro test(c a=);`	Insert the comma between the two parameters, for example, `%macro test(c,a=);`
A close parenthesis is missing from the parameter list when specifying positional parameters.	Add the missing parenthesis to the parameter list.
A positional parameter within the definition contains a special character, for example, `%macro test(a-b,c);`	A parameter name must be a valid SAS name that contains no special characters.

`ERROR: Invalid symbolic variable name value.`

Cause	Solution
The macro variable name contained within a %LOCAL statement or a %GLOBAL statement contains a special character, for example, `%GLOBAL a = b;`	Remove the special character. %LOCAL statements and %GLOBAL statements do not require the macro variable name to have an ampersand. Macro variable names must start with a letter or an underscore and can be followed by letters or digits.

`ERROR: No matching %MACRO statement for this %MEND statement.`

Cause	Solution
The matching %MACRO statement for the %MEND statement is missing.	Add the %MACRO statement. Each %MEND statement must have a matching %MACRO statement.
There is an unclosed comment, missing semicolon, or unmatched parenthesis that is causing the %MACRO statement to not be read.	Close the comment, add the semicolon, or close the parenthesis that appears before the %MACRO statement. After making the correction, you might need to restart your Compute Server session.

`ERROR: Operand missing for value operator in argument to %EVAL function.`

Cause	Solution
The IN operator is being used but there are no values specified within the IN operator, or the operand to the left of the IN operator contains a null value.	When you use the IN operator, both operands must contain a value. If the operand contains a null value, then an error is generated.

`ERROR: No file referenced by SASAUTOS OPTION can be opened.`

Cause	Solution
The SASAUTOS= system option is being used in an OPTIONS statement but the SASAUTOS= system option fileref from the SAS configuration file is missing or commented out.	Ensure that the SASAUTOS= system option fileref is present within the SAS configuration file and that it points to the autocall macro location supplied by SAS.

`ERROR: The text expression value contains a recursive reference to the macro variable value. The macro variable will be assigned the null value.`

Cause	Solution
A macro variable is being assigned to itself. The macro variable did not exist prior to the assignment, for example, `%let a=&a`	Ensure that the macro exists prior to the assignment statement, for example, `%global a; %let a=&a`

`ERROR: Attempt to %GLOBAL a name (value) which exists in a local environment.`

Cause	Solution

A %GLOBAL statement is being used on a macro variable that has already been declared as local, for example,
`%macro test(a);`
`%global a;`
`%mend;`
 	
`%test(100)`
	If a local macro variable needs to be made global, a new global macro variable needs to be created. The new global macro must be equal to the local macro variable, for example,
	`%macro test(a);`
	`%global newa;`
	`%let newa=&a`
	`%mend;`
	
	`%test(100)`

`ERROR: Attempt to assign a value to a read-only symbolic variable (value).`

Cause	Solution
A macro variable is being assigned a value but the macro variable is a read-only SAS automatic macro variable, for example, `%let syserr=0;`	A read-only SAS automatic macro variable cannot be assigned a value. If you are trying to create a new macro variable, then change the name to something that does not match the variables supplied by SAS.

`ERROR: There is no matching %IF statement for the %ELSE.`

Cause	Solution
The %ELSE statement was submitted but there is no %IF statement.	The %ELSE statement must follow a %IF statement.
There is text between the action for the %IF statement and the %ELSE statement. For example, there is an asterisk style comment that is before the %ELSE.	The %ELSE statement must immediately follow the action for the %IF statement. If there is an asterisk style comment before the %ELSE statement, then change it to a PL1 style comment. Here is an example: `/*` *`comment`* `*/`.This style is evaluated at a different time than the asterisk style comment.

`ERROR: There is no matching %DO statement for the %END. This statement will be ignored.`

Cause	Solution
There is a missing %DO statement.	Each %END statement must have a matching %DO statement. If all %END statements have a matching %DO statement, ensure that there is not an unclosed comment or missing semicolon prior to the %DO statement.

`ERROR: There is no matching %IF statement for the %THEN.`

Cause	Solution
There is a missing %IF statement.	Each %THEN statement must have a matching %IF statement. If each %THEN statement has a matching %IF statement, then ensure that there is not an unclosed comment or missing semicolon prior to the %IF statement.

`ERROR: Nesting of %IF statements in open code is not supported. %IF ignored.`

Cause	Solution
%IF statements were nested in open code.	Put the nested %IF statements in a macro definition.

ERROR: The %DO statement is not valid in open code.

ERROR: The %END statement is not valid in open code.

ERROR: The %LOCAL statement is not valid in open code.

ERROR: The %GOTO statement is not valid in open code.

ERROR: The %ABORT statement is not valid in open code.

ERROR: The %RETURN statement is not valid in open code.

Cause	Solution
These statements are being executed outside a macro definition.	Except for %DO and %END, all these statements must reside between a %MACRO statement and a %MEND statement. In open code, %DO and %END must be used only with %IF-%THEN/ELSE statements. Otherwise, they must reside between a %MACRO statement and a %MEND statement.
These statements are contained within a file that is being included using the %INCLUDE statement.	Statements within a %INCLUDE statement that are within a macro are not executed within the macro. The lines of code with the %INCLUDE statement must be stand-alone code.
There is an unclosed comment, unmatched quotation mark, or missing semicolon prior to the %MACRO statement.	Close all comments, match all quotation marks, and ensure that all statements have a semicolon. After making the correction, you might need to restart your Compute Server session.

ERROR: The %GOTO statement has no target. The statement will be ignored.

Cause	Solution
A macro variable was used as the label in a %GOTO statement but has a null value.	The label for the %GOTO statement must be a valid SAS name.

ERROR: The macro value contains a %GOTO statement with an invalid statement label name. The macro will not be compiled.

Cause	Solution
A %GOTO statement points to a label that does not exist.	Each %GOTO statement must have a valid label statement that begins with a percent sign and is followed by a colon.
There is an unclosed comment, unmatched quotation mark, or missing semicolon prior to the %LABEL statement.	Close all comments, match all quotation marks, and ensure that all statements have a semicolon. After making the correction, you might need to restart your Compute Server session.
The label is not a valid SAS name, for example, `%a-1:`	The label must be a valid SAS name that contains no special characters.

`ERROR: In macro value, the target of the statement %GOTO value, resolved into the label value, which was not found.`

Cause	Solution
A percent sign precedes the label name of the %GOTO statement, for example, `%goto %a;`	Remove the percent sign. If you want to invoke a macro and return a label name, then ensure that the macro contains only a valid label name with no semicolon.
An ampersand precedes the label name of the %GOTO statement, for example, `%goto &a;`	Ensure that the macro variable exists and that it returns a valid label name value.

`ERROR: In macro value, the target of the statement %GOTO value, resolved into the label value, which is not a valid statement label.`

Cause	Solution
The label is a macro or macro variable that contains a character that is not valid in a label name.	Ensure that the macro invocation or macro variable returns a valid SAS name as a label. If the label name is from a macro invocation, then ensure that there is no semicolon following the label name.
The %GOTO statement contains a label that is not a valid SAS name, for example, `%goto a-1;`	Ensure that the %GOTO statement contains a valid SAS name for the label and that it does not contain special characters.

`ERROR: value is an invalid macro variable name for the index variable of the %DO loop.`

Cause	Solution
The index variable is not a valid SAS name, for example, `%do 1a=1 %to 3;`	The index variable must be a valid SAS name, which starts with a letter or an underscore and can be followed by letters or digits, for example, `%do a1=1 %to 3;`
The index variable contains an ampersand but there is no macro variable with that name.	Remove the ampersand, or create the macro variable first that contains a valid SAS name.
The index variable contains an ampersand but the macro variable resolves to a null value or invalid SAS name.	Ensure that the macro variable that resolves as the index variable resolves to a valid SAS name.

`ERROR: The value value of the %DO value loop is invalid.`

Cause	Solution
Either the FROM or TO value is not an integer value.	The FROM and TO values must be an integer or a macro expression that generates integers.

`ERROR: The %BY value of the %DO value loop is zero.`

Cause	Solution
The value of the %BY statement is zero, or a macro variable is used and resolves to zero.	The value of the %BY statement must be an integer (other than zero) or a macro expression that resolves to an integer.

`ERROR: Expected %TO not found in %DO statement.`

Cause	Solution
The %TO statement is missing from an iterative %DO loop, for example, `%do i=1 3;`	An iterative %DO loop must contain a %TO statement, for example, `%do i=1 %to 3;`

`ERROR: Invalid macro parameter name value. It should be a valid SAS identifier no longer than 32 characters.`

Cause	Solution
The parameter name within a macro definition contains an ampersand.	Remove the ampersand. Parameter names that are specified within a macro definition cannot contain an ampersand.
The parameter name that is within a macro definition contains a percent sign.	Remove the percent sign. Parameter names that are specified within a macro definition cannot contain a percent sign.
The parameter name is longer than 32 characters.	A macro variable name must be 32 characters or less.

`ERROR: Expected equal sign not found in value statement.`

Cause	Solution
A %LET statement within a macro definition is missing the equal sign.	The %LET statement must contain an equal sign that follows the macro variable name.
A %SYSLPUT statement or a %SYSRPUT statement is within a macro definition and is missing an equal sign.	The %SYSLPUT statement and the %SYSRPUT statement must contain an equal sign that is between the macro variable name and the value.
There is a missing equal sign on a %GLOBAL or %LOCAL statement while trying to create a read-only macro variable, for example, `%global / readonly newtest;`	Include the equal sign when creating a read-only macro variable, for example, `%global / readonly newtest=100;`

`ERROR: An unexpected semicolon occurred in the %DO statement.`

Cause	Solution
There is a missing equal sign that follows the index variable in the %DO statement.	Place an equal sign after the index variable name, for example, `%do i=1 %to 3;`

`ERROR: %EVAL function has no expression to evaluate, or %IF statement has no condition.`

Cause	Solution
An %IF statement does not contain an expression to evaluate before the %THEN statement.	Add an expression to evaluate between the %IF statement and the %THEN statement.
A %DO %UNTIL statement or a %DO %WHILE statement does not contain any text between the parentheses.	A %DO statement requires an expression to evaluate.
The %EVAL function does not contain any text between the parentheses.	A %EVAL function requires an expression to evaluate.
A function that is being used to generate an expression returns a null value, for example, `%if %eval(a) %then`	Ensure that the function returns a valid expression.

`ERROR: Required operator not found in expression: value`

Cause	Solution
There is an extra close parenthesis within the macro function that is being used.	Remove the extra close parenthesis.
There is a colon following a parenthesis within a %IF statement.	If a macro variable contains a colon, then use the %SUPERQ function around the macro variable, or use the %STR function around the text.
A SAS function is being used on a macro statement, such as the %IF statement.	Use the %SYSFUNC function around the SAS function.
The IN operator is being used within the macro facility.	The MINOPERATOR system option must be set before using the IN operator.
A macro is being invoked from a statement, such as a %IF statement, but the macro does not exist.	Ensure that the macro that is invoked has been compiled. If it is an autocall macro, then ensure that the SASAUTOS= system option is pointing to the location of the macro.

`ERROR: Macro function value has too many arguments.`

Cause	Solution
The function being used requires a certain number of arguments and that number has been exceeded. Here is an example: `%substr(abcd,1,2,3)`.The %SUBSTR function can handle only three arguments.	Ensure that the required number of arguments are met and not exceeded. If the text string contains a comma, then use the %STR function around the text.
A macro variable is being referenced as an argument and it contains a comma.	Use a quoting function around the macro variable, such as the %BQUOTE function, to mask the comma.

`ERROR: Macro function value has too few arguments.`

Cause	Solution
The function being used requires a certain number of arguments and that number has not been reached. Here is an example: `%substr(abcd)`.The %SUBSTR function requires at least two arguments.	Ensure that the required number of arguments are met.

`ERROR: There were value unclosed %DO statements. The macro value will not be compiled.`

Cause	Solution
There is a missing %END statement for a %DO statement.	Every %DO statement requires a corresponding %END statement.
There is a missing semicolon, unclosed comment, or unmatched quotation mark prior to a %END statement.	Ensure that all comments are closed, each statement has a semicolon, and each quotation mark has a matching quotation mark prior to the %END statement. After making the correction, you might need to restart the Compute Server session.

`ERROR: Argument value to macro function value is not a number.`

Cause	Solution
The second argument of the %SCAN function or the %QSCAN function is not an integer, instead it is a character value.	The second argument of the %SCAN function or the %QSCAN function must be a number or an expression that resolves to an integer.
The second or third argument of the %SUBSTR function or the %QSUBSTR	The second and third arguments to the %SUBSTR function or the %QSUBSTR

Cause	Solution
function is not an integer, instead it is a character value.	function must be a number or an expression that resolves to an integer.

`ERROR: The condition in the %DO value loop, value, yielded an invalid or missing value, value. The macro will stop executing.`

Cause	Solution
A condition within the %DO %UNTIL statement or the %DO %WHILE statement resolves to a null value or nonnumeric characters.	The macro expression within the %DO %UNTIL statement or the %DO %WHILE statement must resolve to a logical value. The expression is true if it is an integer other than zero. The expression is false if it has a value of zero.

`ERROR: Invalid branch into iterative %DO.`

Cause	Solution
The label for a %GOTO statement is contained within a %DO and %END statement.	Move the label outside of the iterative %DO loop.

`ERROR: Division by zero in %EVAL is invalid.`

Cause	Solution
The denominator of a calculation within a %EVAL function is zero.	Change the denominator to something other than 0.
An expression within a %IF or %DO loop contains a calculation with 0 as the denominator.	Ensure that the expression does not contain a calculation that is dividing by 0.

`ERROR: The keyword parameter value was not defined with the macro.`

Cause	Solution
The keyword parameter specified in the macro invocation does not exist in the definition.	Ensure that each parameter that is used in the invocation also exists in the definition.
The parameter value contains an equal sign.	Use the %STR function around the value in the macro invocation, for example, `%test(%str(a=100))`

`ERROR: The macro variable name is either all blank or missing.`

Cause	Solution
There is no text between the quotation marks of the first argument within the CALL SYMPUTX routine or the CALL SYMPUT routine.	A valid SAS name must be specified in the first argument to create the macro variable name.
The DATA step variable being used as the macro variable name within the CALL SYMPUT routine or the CALL SYMPUTX routine has no value.	Ensure that the DATA step variable contains a valid SAS name.

`ERROR: Expected %THEN statement not found.`

Cause	Solution
A %IF statement does not contain a %THEN statement following the expression.	Add the %THEN statement following the expression within the %IF statement.

`ERROR: Overflow has occurred; evaluation is terminated.`

Cause	Solution
The value within the %SYSEVALF function or the %EVAL function has gone beyond 1.79e308.	Ensure that the value is less than 1.79e308 or add quotation marks around the value so that it is treated as a text value.

`ERROR: %GO not followed by TO.`

Cause	Solution
The TO portion of the %GOTO statement is missing.	Ensure that the %GOTO statement is spelled correctly.

`ERROR: The target of the %GOTO statement was a reserved macro keyword, value.`

Cause	Solution
The label portion of the %GOTO statement is a macro reserved word.	Change the %GOTO label so that it does not match any reserved words. See Appendix 1, " Reserved Words in the Macro Facility," on page 483.

`ERROR: All positional parameters must precede keyword parameters.`

Cause	Solution
The parameters followed by an equal sign precedes parameters without an equal sign in the definition.	If using keyword (with equal sign) parameters and positional (without equal sign) parameters together, then all positional parameters must come before the keyword parameters.
The parameter value contains a comma that is meant to be text to the parameter.	Mask the string being passed to the macro with the %STR function, for example, `%test(1,a=%str(4,f))`

`ERROR: The keyword parameter value passed to macro value was given a value twice.`

Cause	Solution
The same parameter name is listed more than once within a macro invocation.	Each parameter name can appear only once within the parameter list.

`ERROR: The index variable in the %DO value loop has taken on an invalid or missing value. The macro will stop executing.`

Cause	Solution
The index variable of a macro %DO statement has been set to missing or given a non-numeric value within the loop.	If there is a nested macro invocation within the loop, then ensure that the %DO index variable is not getting reset in the nested macro. Either change the index variable name or change the matching name within the nested macro.

`ERROR: Literal contains unmatched quote.`

Cause	Solution
A value contains a missing or extra quotation mark or an apostrophe.	Match the missing quotation mark. If the unmatched quotation mark or apostrophe is part of the text, then a macro quoting function, such as %STR, is needed, for example, `%str(joe%'s diner)` Precede the apostrophe or unmatched quotation mark with a percent sign.

`ERROR: The SAS System was unable to open the macro library.`

Cause	Solution
A stored compiled macro was moved from one release to another or from one operating system to another.	A stored compiled macro cannot be moved to a different operating system or to a different release of SAS. The macro must be recompiled in the new location or release.
There is a leftover file within the Work directory that you might not have correct permissions for use.	Clean out the Work directory. The following SAS Note might help: http://support.sas.com/kb/8/786.html.
NOWORKINIT system option is set.	Run the job with the WORKINIT system option set.

`ERROR: The SAS System is unable to write the macro value to the macro library.`

Cause	Solution
An attempt was made to compile a macro to a permanent location that is specified by the SASMSTORE= system option.	The SASMACR catalog that is being written to is either corrupt or the catalog contains stored compiled macros from a different SAS release or different operating system. Write to a new location.

`ERROR: Expected semicolon not found after value clause.`

Cause	Solution
There is a macro variable reference on the macro name of a definition, for example, `%macro test&i;`	A macro name must be a valid SAS name. You cannot use a text expression to generate a macro name in a %MACRO statement.
The word %MACRO is repeated before a macro invocation.	When calling or invoking a macro you do not include the word %MACRO before it.
A macro variable is being referenced on the macro definition line to declare options.	A macro variable cannot be used to generate options on a macro definition. Remove the macro variable and hardcode the options that you need.

`ERROR: Open code statement recursion detected.`

Cause	Solution

A statement within a macro is missing a semicolon.	Ensure that every statement that requires a semicolon has one. Ensure that there is not an unclosed comment that is causing a semicolon to not be seen. Run the following code after you make the correction:
	`*'; *"; *); */; %mend;`
	`run;`

A function is missing a parenthesis.	Ensure that all parenthesis match. Run the following code after you make the correction:
	`*'; *"; *); */; %mend;`
	`run;`

A macro variable is being referenced in a function that might contain unmatched quotation marks, parenthesis, or a comment.	Use a macro quoting function around the macro variable, such as the %SUPERQ function. If you are using a function, such as the %SUBSTR function, then switch to %QSUBSTR.
A comment, such as `*`, is being used inside a macro.	Do not use `*` style comments inside a macro. The PL1 style comments `/* */` should be used inside a macro.
Quotation marks are being used around a macro variable in a function, such as %SUBSTR, for example, `%let temp=%substr("abc",1,2);`	Remove the quotation marks from around the macro variable.

```
ERROR: A dummy macro will be compiled.
```

Cause	Solution
If the macro processor detects a syntax error while compiling the macro, then it checks the syntax in the rest of the macro. Messages are issued for any additional errors that are found. However, the macro processor does not store the macro for execution. A macro that is compiled by the macro processor but is not stored is called a *dummy macro*.	Correct the syntax and compile the macro again.

```
ERROR: Macro keyword value is not yet implemented.
```

Cause	Solution
The keyword referenced in the error message is a functionality of the macro facility but is not yet available for use.	Ensure that you are running the correct release of SAS that contains this functionality.

`ERROR: Macro keyword value appears as text.`

Cause	Solution
There is a missing semicolon in a line prior to the statement being flagged with the error message.	Add the missing semicolon. If you are running interactively, then SAS might need to be restarted.
There is an unmatched parenthesis before the statement that is being flagged with the error message.	Add the missing parenthesis. If you are running interactively, then SAS might need to be restarted.
A macro statement is being used incorrectly within a %LET statement within a macro definition, for example, `%let x= %put test;`	Macro statements cannot be used inside a %LET statement. Change the example %LET statement to the following: `%let x=%nrstr(%put test;);`

`ERROR: The macro name value and the internal macro header name value in the SASMACR macro catalog differ. Recompile this macro and try again.`

Cause	Solution
There is an extra or missing parenthesis on the macro invocation.	Ensure that the parentheses have matching pairs. You might need to restart the Compute Server session.
There is an unmatched parenthesis, quotation mark, or a comment within the macro.	Ensure that there is a matching pair of parentheses, quotation marks, or a comment. You might need to restart the Compute Server session.
There is a partial nested macro definition with the same name as the outer macro.	Remove the nested macro definition. If nesting is needed, then ensure that the macro names are different. The best practice is to nest the macro invocation, not the macro definition.

`ERROR: Macro library damaged. Cannot compile macro value.`

Cause	Solution
A compiled macro was copied to a different operating system or release of SAS than where it was compiled.	Compiled macro catalog entries can be executed only on the same release of SAS and on the same operating system where they were created. Moving compiled macros across operating systems or releases of SAS is not supported. The macros need to be recompiled on the new operating system or new release of SAS.
A compiled macro is being permanently stored in a SASMACR catalog that contains macros from a different operating system or release of SAS.	Compile the macro to a new location to create a new SASMACR catalog.

`ERROR: Unable to load KEYS for the window value.`

Cause	Solution
Setting the KEYS= option to a value that does not exist.	The value of the KEYS= option must be a valid *libref.catalog* combination.

`ERROR: %SYSRPUT statement is valid only when OPTION DMR is in effect.`

Cause	Solution
The NODMR system option is set when SAS starts up and an attempt was made to use the %SYSRPUT function.	The DMR system option must be set in order to be able to use %SYSRPUT.

`ERROR: Macro value has been given a reserved name.`

Cause	Solution
An attempt was made to define a macro that has the same name as a macro function or macro statement. (This does not include the autocall macros that are supplied by SAS.)	When you name a user-defined macro, follow the rules for naming macros.See Appendix 1, " Reserved Words in the Macro Facility," on page 483.

`ERROR: Invalid branch into %DO %WHILE.`

Cause	Solution
An attempt was made to use %GOTO logic within a %DO %WHILE loop.	%GOTO syntax cannot be used within a %DO %WHILE loop. Remove it from the loop.

`ERROR: Invalid branch into %DO %UNTIL.`

Cause	Solution
An attempt was made to use %GOTO logic within a %DO %UNTIL loop.	%GOTO syntax cannot be used within a %DO %UNTIL loop. Remove it from the loop.

`ERROR: Extraneous text on %MACRO statement ignored.`

Cause	Solution
The macro statement contains a slash (/) that is followed by an invalid option.	Only valid options can be used in the %MACRO statement. The following is a list of valid options: CMD DES= MINDELIMITER= MINOPERATOR PARMBUF SECURE | NOSECURE STMT SOURCE | SRC STORE

`ERROR: The MSTORED option must be set to use the /STORE macro statement option.`

Cause	Solution
The /STORE option is used in the %MACRO statement and the MSTORED system option is not set.	List the MSTORED system option in an OPTIONS statement before using the STORE option in the %MACRO statement.

`ERROR: The syntax for this %MACRO statement option is /DES = "description".`

Cause	Solution
The description used with the DES= option is not within quotation marks.	The value of the DES= option in the %MACRO statement must be within quotation marks.

ERROR: The WORK.SASMACR catalog is temporary and can not be used for compiled stored macros. Change OPTION SASMSTORE to a different libref.

Note: The Work.Sasmacr catalog is used to store compiled macros for the primary SAS session. In applications or programs that use side sessions, the catalog used to store compiled macros for each side session is Work.Sasmac*n*, where *n* is a unique integer.

Cause	Solution
The SASMSTORE= option is set to the Work library.	Set the SASMSTORE= option to a valid library other than the Work library.

ERROR: The option SASMSTORE = libref is not set.

Cause	Solution
The MSTORED system option is set and the macro is defined with the /STORE option without setting the SASMSTORE= system option.	Set the SASMSTORE= system option to a valid library when using the STORE option in the %MACRO statement.

ERROR: Expected close parenthesis after macro function invocation not found.

Cause	Solution
The close parenthesis is omitted when using a macro function.	The syntax for every macro function requires an open and close parenthesis.

ERROR: Macro parameter contains syntax error.

Cause	Solution
An attempt was made to invoke a macro with CALL EXECUTE when a parameter value has an	The unmatched parenthesis in the parameter must be masked with the %STR function and preceded by a percent sign. The following is an example: `call=cats('%test(%str`

Cause	Solution

open parenthesis and not a close parenthesis.	`(',tranwrd(tranwrd(x,`
	`'(','%('),')','%)'),')))');`
	`call execute(call);`

`ERROR: The value function referenced in the %SYSFUNC or %QSYSFUNC macro function is not found.`

Cause	Solution
Listing a function within %SYSFUNC that does not exist.	Check the function documentation to make sure that the first argument to the %SYSFUNC function is valid.
An attempt was made to use the PUT or INPUT function with the %SYSFUNC function.	Use the INPUTC, INPUTN, PUTC, or PUTN function instead of the PUT or INPUT function.

`ERROR: The function value referenced by the %SYSFUNC or %QSYSFUNC macro function has too many arguments.`

Cause	Solution
More commas are found in the syntax of the function than there are arguments to that function. The most likely cause is that an argument to the function is a macro variable and the resolved value of the macro variable contains commas.	Ensure that the correct syntax is used for the function. If the comma or commas causing the error are within the resolved value of the macro variable that is used with the function, then that variable must be masked using the %BQUOTE function.

`ERROR: The function value referenced by the %SYSFUNC or %QSYSFUNC macro function has too few arguments.`

Cause	Solution
Not all of the required arguments are listed in the function's syntax.	Refer to the documentation for the function mentioned in the error and ensure that all required arguments are listed.

`ERROR: Argument value to function value referenced by the %SYSFUNC or %QSYSFUNC macro function is not a number.`

Cause	Solution
A nonnumeric argument value is used instead of the expected numeric value.	Refer to the documentation for the function mentioned in the error. Ensure that numeric values are used for the arguments that require numeric values.

```
ERROR: %SYSEVALF must be followed by an expression enclosed in
parentheses.
```

Cause	Solution
The expression to be evaluated is not within parentheses.	Enclose the expression to be evaluated in parentheses.

```
ERROR: The function value referenced by %SYSFUNC, %QSYSFUNC, or
%SYSCALL cannot be used within the MACRO function/call-routine
interfaces.
```

Cause	Solution
An unapproved function is used as the first argument to the %SYSFUNC function.	The following variable information functions are not available for use with the %SYSFUNC function: . ALLCOMB LEXCOMBI ALLPERM LEXPERK DIF LEXPERM DIM LEXCOMB HBOUND MISSING IORCMSG PUT INPUT RESOLVE LAG SYMGET LBOUND

```
ERROR: Unknown %SYSEVALF conversion operand value specified;
conversion is terminated.
```

Cause	Solution
The conversion type specified in the %SYSEVALF syntax is invalid.	The following valid values are for the second argument to the %SYSEVALF function: BOOLEAN FLOOR CEIL INTEGER

```
ERROR: The %SYSEVALF ROUND conversion operation is not supported.
```

Cause	Solution
An attempt was made to use ‘ROUND’ as the conversion type in the %SYSEVALF syntax.	The following valid values are for the second argument to the %SYSEVALF function: BOOLEAN FLOOR CEIL INTEGER

ERROR: %SYSEVALF detected a missing value during the conversion operation requested; conversion is terminated.

Cause	Solution
There is a missing value within the expression being evaluated by the %SYSEVALF function and a conversion type is being used.	Use only nonmissing values within the first argument to the %SYSEVAL function.

ERROR: %SYSEVALF function has no expression to evaluate.

Cause	Solution
The first argument to the %SYSEVALF function is omitted.	Use only nonmissing values within the first argument to the %SYSEVALF function. See “No Expression to Evaluate” on page 354.

ERROR: The %SYSFUNC or %QSYSFUNC macro function has too many arguments. The excess arguments will be ignored.

Cause	Solution
More commas are found in the syntax than expected. The most common cause of this error is an argument to the %SYSFUNC function that contains unmasked commas.	Ensure that the correct number of commas is used within the syntax. If an argument to a function within the %SYSFUNC function contains commas, then the commas must be masked using the %BQUOTE function.

ERROR: Invalid arguments detected in %SYSCALL, %SYSFUNC, or %QSYSFUNC argument list. Execution of %SYSCALL statement or %SYSFUNC or %QSYSFUNC function reference is terminated.

Cause	Solution
There is an invalid argument to the function being used within the %SYSFUNC function.	Check the syntax for the function that is being used with the %SYSFUNC function to make sure that valid arguments are being used.

ERROR: Format name value not found or the width and/or decimal specified for the format used are out of range.

Cause	Solution
This occurs when you specify the format as the second argument to the %SYSFUNC function. This occurs when you use an unknown format, an invalid width, or both, for a format within the %SYSFUNC function.	Ensure that the format used as the optional second argument to the %SYSFUNC function is valid.

ERROR: Expecting a variable name after %LET.

Cause	Solution
A null value is between the %LET statement and the equal sign (=).	A valid macro variable name must immediately follow the %LET statement.

ERROR: Expecting a variable name after %SYSRPUT.

Cause	Solution
A null value is between the %SYSRPUT statement and the equal sign (=).	A valid macro variable name must immediately follow the %SYSRPUT statement.

ERROR: Expected equal sign not found in %LET statement.

Cause	Solution
The equal sign (=) in the %LET statement syntax was omitted. This error occurs only within a macro definition.	Ensure that the syntax for the %LET statement is correct. `%LET` *`variable-name=value;`*

ERROR: Expected equal sign not found in %SYSRPUT statement.

Cause	Solution
A valid macro variable name does not follow %SYSRPUT statement.	A macro variable name must follow these rules: ■ SAS macro variable names can be up to 32 characters in length. ■ The first character must begin with a letter or an underscore. Subsequent characters can be letters, numeric digits, or underscores.

Cause	Solution
	■ A macro variable name cannot contain blanks. ■ A macro variable name cannot contain double-byte character set (DBCS) characters. ■ A macro variable name cannot contain any special characters other than the underscore. ■ Macro variable names are case insensitive. For example, cat, Cat, and CAT all represent the same variable. ■ You can assign any name to a macro variable as long as the name is not a reserved word. The prefixes AF, DMS, SQL, and SYS are not recommended because they are frequently used in SAS software for automatic macro variables. Thus, using one of these prefixes can cause a name conflict with an automatic macro variable. For a complete list of reserved words in the macro language, see Appendix 1, " Reserved Words in the Macro Facility," on page 483. If you assign a macro variable name that is not valid, an error message is printed in the SAS log.
A macro variable that does not exist has been listed after the %SYSRPUT statement.	Review the possible reasons for a macro variable not resolving.

ERROR: The %SYSEVALF macro function has too many arguments. The excess arguments will be ignored.

Cause	Solution
The %SYSEVALF function has only two possible arguments. The presence of more than one comma produces this error. An extra comma is often found in the resolved value of a macro variable.	Omit any commas when listing the arguments to the %SYSEVALF function. If the argument to %SYSEVALF is a macro variable that contains a comma, then the comma can be removed using the COMPRESS function. `%sysevalf( %sysfunc(compress(%bquote(&x), %str(,)))*&y)`

ERROR: Function name missing in %SYSFUNC or %QSYSFUNC macro function reference.

Cause	Solution
No function is listed in the %SYSFUNC syntax.	The first argument to the %SYSFUNC function must be a valid SAS function.

`ERROR: CALL routine name missing in %SYSCALL macro statement.`

Cause	Solution
The call routine name after the %SYSCALL statement was omitted.	A valid call routine name must immediately follow %SYSCALL.

`ERROR: Macro variable name value must start with a letter or underscore.`

Cause	Solution
An attempt was made to reference an invalid macro variable name in the %SYMDEL statement.	A macro variable name must follow these naming conventions: ■ SAS macro variable names can be up to 32 characters in length. ■ The first character must begin with a letter or an underscore. Subsequent characters can be letters, numeric digits, or underscores. ■ A macro variable name cannot contain blanks. ■ A macro variable name cannot contain double-byte character set (DBCS) characters. ■ A macro variable name cannot contain any special characters other than the underscore. ■ Macro variable names are case insensitive. For example, cat, Cat, and CAT all represent the same variable. ■ You can assign any name to a macro variable as long as the name is not a reserved word. The prefixes AF, DMS, SQL, and SYS are not recommended because they are frequently used in SAS software for automatic macro variables. Thus, using one of these prefixes can cause a name conflict with an automatic macro variable. For a complete list of reserved words in the macro language, see Appendix 1, " Reserved Words in the Macro Facility," on page 483. If you assign a macro variable name that is not valid, an error message is printed in the SAS log.

`ERROR: Compiled stored macro value has was invoked using statement-style invocation. This compiled stored macro was not compiled as a statement-style macro. Execution of this compiled stored macro ends. Use name-style invocation for this compiled stored macro.`

Cause	Solution
An attempt was made to invoke a macro as a statement-style macro when the macro was not defined as a statement-style macro. This error message is usually issued when the IMPLMAC system option is set and the percent sign (%) that should precede the macro invocation is omitted.	Ensure that the macro invocation begins with a percent sign (%). When you use the IMPLMAC system option, the processing time is increased because SAS searches the macros that are compiled during the current session for a name corresponding to the first word of each SAS statement. If you are not using statement-style macros, then make sure that the NOIMPLMAC system option is set.

ERROR: Compiled stored macro value has was invoked using command-style invocation. This compiled stored macro was not compiled as a command-style macro. Execution of this compiled stored macro ends. Use name-style invocation for this compiled stored macro.

Cause	Solution
An attempt was made to invoke a macro as a command-style macro when the macro was not defined as a command-style macro. This error message is usually generated when attempting to invoke a macro on the command line.	Invoke the macro within the body of the Code Editor.

ERROR: WORK is at level 1 of the concatenated libref value for compiled stored macros value. Change OPTION SASMSTORE to a different libref without WORK at level 1. WORK is temporary.

Cause	Solution
An attempt was made to use a libref that is the concatenation of a library and the Work library as the value of the SASMSTORE= system option. The Work library is listed first in the LIBNAME statement.	Do not point to the Work library when assigning the value of the SASMSTORE= system option. Work is a temporary library and is not a possible location for a permanent macro catalog.

ERROR: Expecting a variable name after %SYSLPUT.

Cause	Solution
A valid macro variable name does not follow the %SYSLPUT statement.	Ensure the text that immediately follows the %SYSLPUT statement is a valid macro variable name. If the %SYSLPUT statement is followed by a macro variable reference, ensure that the resolved value of that macro variable is a valid macro variable name.

ERROR: Expected equal sign not found in %SYSLPUT statement.

Cause	Solution
An equal sign (=) that should appear between the macro variable that is being created and its value has been omitted. This error message occurs only within a macro.	An equal sign (=) must separate the macro variable that is being created and its value.

ERROR: The value call routine referenced in the %SYSCALL macro statement is not found.

Cause	Solution
A non-existent call routine was listed.	A valid call routine must be listed. All SAS call routines are accessible with the %SYSCALL statement except LABEL, VNAME, SYMPUT, and EXECUTE.

ERROR: Improper use of macro reserved word value.

Cause	Solution
An attempt was made to use a reserved macro word incorrectly.	The correct syntax must be used when using macro syntax such as the %TO, %BY, and %THEN statements.

ERROR: Attempt to delete automatic macro variable value.

Cause	Solution
An attempt was made to delete a macro variable defined by SAS.	Only user-defined global macro variables can be deleted using the %SYMDEL statement.

ERROR: Unrecognized option to the %SYSLPUT statement.

Cause	Solution
An invalid option is listed in the %SYSLPUT statement.	REMOTE= is the only valid option in the %SYSLPUT statement.

ERROR: The text expression length (value) exceeds maximum length (value). The text expression has been truncated to value characters.

Cause	Solution
A text expression within the macro language exceeds 65534 bytes. For example, this error message is generated if the argument to a macro function exceeds 65534 bytes.	A macro variable value cannot exceed 65534 bytes.

ERROR: Unrecognized option on %ABORT statement: value

Cause	Solution
An unknown option is used in the %ABORT statement.	Valid options in the %ABORT statement are: ABEND, CANCEL, EXIT, and RETURN.

ERROR: Execution terminated by an %ABORT statement.

Cause	Solution
This error is generated when the %ABORT statement executes successfully.	This error is expected when the %ABORT statement is executed.

ERROR: The MSTORED option must be set to use the /SOURCE macro statement option.

Cause	Solution
The /SOURCE option is used in the %MACRO statement and the MSTORED system option has not been used.	The MSTORED and SASMSTORE= system options must be used in order to use the stored compiled macro facility.

ERROR: Extraneous text on %COPY statement ignored.

Cause	Solution

Text that comes after the slash (/) in the %COPY statement is not a valid option.	Valid options for the %COPY statement are:
	`LIBRARY= | LIB=`
	`OUTFILE= | OUT=`

ERROR: The syntax for this %COPY statement option is /LIBRARY = libref.

Cause	Solution
The equal sign (=) that should follow the LIBRARY option is omitted.	The following is the correct syntax for the LIBRARY= option: /library=*valid-libref*

ERROR: The syntax for this %COPY statement option is /OUTFILE = <fileref> | "filename".

Cause	Solution
The equal sign (=) that should follow OUTFILE is omitted.	The following is the correct syntax for the OUTFILE= option: `/outfile=`*`fileref`*`|`*`'external file name'`*

ERROR: This combination of default and specified %COPY statement options is not supported.

Cause	Solution
At least one option is listed in the %COPY statement, but the SOURCE option is omitted.	The SOURCE option must be listed if using the %COPY statement.

ERROR: Macro value not found in libref value.

Cause	Solution
The macro listed in the %COPY statement is not found in the libref listed on the LIBRARY= option.	Ensure that the macro listed in the %COPY statement is stored in the Sasmacr.sas7bcat catalog located in the library referenced by the SASMSTORE= system option.

ERROR: The /SOURCE option was not specified when the macro value was compiled.

Cause	Solution
The macro listed in the %COPY statement was not compiled with the SOURCE option set.	Only stored compiled macros that were compiled with the SOURCE option can be listed in the %COPY statement.

ERROR: An error occurred during the execution of the %COPY statement.

Cause	Solution
This error message follows any syntax error that is generated by the %COPY statement.	The following is the correct syntax for the %COPY statement: `%copy macro-name /<option-1 <option-2> ...> source`

```
ERROR: Invalid or missing macro name specified in the %COPY
statement, value.
```

Cause	Solution
The macro name is omitted or the macro name is not a valid macro name.	Ensure that the macro name listed in the %COPY statement exists in the Sasmacr.sas7bcat catalog located in the library referenced by the SASMSTORE= option.

```
ERROR: The /SOURCE option cannot be used on a macro definition
enclosed within another macro.
```

Cause	Solution
A macro definition nested within another macro definition has the SOURCE option listed in addition to the STORE option.	A nested macro definition cannot be defined as a stored compiled macro. Nested macro definitions are not recommended because of the decrease in efficiency.

```
ERROR: The /SOURCE option cannot be used without the /STORE option.
```

Cause	Solution
The SOURCE option is listed without listing the STORE option.	The SOURCE option is valid only with the STORE option.

```
ERROR: Conflicting use of /SECURE and /NOSECURE options.
```

Cause	Solution
The SECURE and NOSECURE options are listed simultaneously in the %MACRO statement.	Specify only the SECURE option or the NOSECURE option in the %MACRO statement.

```
ERROR: The /MINDELIMITER= option must be a single character enclosed
in single quotation marks.
```

Cause	Solution
More than one character is listed as the delimiter.	Only one character can be listed as the delimiter.
The value for the MINDELIMITER= option is listed within double quotation marks.	The character must be within single quotation marks.

ERROR: The /MINDELIMITER= option can only be specified once.

Cause	Solution
The MINDELIMITER= option is listed more than once in the %MACRO statement.	Only one delimiter can be used. The MINDELIMITER= option can be specified only once.

ERROR: Attempt to define more than one parameter with same name: value.

Cause	Solution
An attempt was made to list multiple parameters with the same name.	Each macro parameter must have a unique name.

ERROR: Execution canceled by an %ABORT CANCEL statement.

Cause	Solution
This error message is generated when the %ABORT statement is executed.	This error message simply indicates that the %ABORT statement executed.

ERROR: Execution canceled by an %ABORT CANCEL FILE statement.

Cause	Solution
The %ABORT CANCEL statement is executed and the FILE option is specified.	This error message simply indicates that the %ABORT CANCEL statement executed and that the FILE option has been specified.

ERROR: Attempt to %LOCAL automatic macro variable value.

Cause	Solution
A macro variable defined by SAS is listed in the %LOCAL statement.	List only user-defined macro variables in the %LOCAL statement.

`ERROR: Attempt to %GLOBAL automatic macro variable value.`

Cause	Solution
A macro variable defined by SAS is listed in the %GLOBAL statement.	List only user-defined macro variables in the %GLOBAL statement.

`ERROR: Conflicting use of /MINOPERATOR and /NOMINOPERATOR options.`

Cause	Solution
The MINOPERATOR and NOMINOPERATOR options are listed on the same %MACRO statement.	List only the MINOPERATOR option in the %MACRO statement if the macro IN operator is going to be used within the macro that is being defined. List only the NOMINOPERATOR option if the MINOPERATOR system option has been set. The macro IN operator should not be available in the macro that is being defined.

`ERROR: Attempt to execute the /SECURE macro value within a %PUT statement.`

Cause	Solution
A macro defined with the SECURE option is invoked in a %PUT statement.	A macro defined with the SECURE option cannot be executed in a %PUT statement.

`ERROR: The macro value is still executing and cannot be redefined.`

Cause	Solution
If a macro is invoked and a parameter has an open parenthesis with no closing parenthesis, then the macro processor looks for the closing parenthesis. If an attempt is made to invoke the same macro, this error is issued.	The macro parameter list must have an open and close parentheses.

`ERROR: The value supplied for assignment to the numeric automatic macro variable SYSCC was out of range or could not be converted to a numeric value.`

Cause	Solution
An attempt was made to use a value of 9999999999 or greater for the SYSCC automatic macro variable.	The value for the SYSCC automatic macro variable must be less than 9999999999.

ERROR: The variable value was previously declared and cannot be made READONLY.

Cause	Solution
An attempt was made to list an existing macro variable on a %GLOBAL or %LOCAL statement with the READONLY option specified.	An existing macro variable cannot be redefined as read-only. A new macro variable must be created.

ERROR: The variable value was declared READONLY and cannot be modified or re-declared.

Cause	Solution
An attempt was made to redefine a macro variable that was created with the READONLY option on a %GLOBAL or %LOCAL statement.	A read-only macro variable cannot be redefined.

ERROR: The variable value was declared READONLY and cannot be deleted.

Cause	Solution
A macro variable defined as read-only has been listed in the %SYMDEL statement.	Read-only macro variables cannot be deleted using the %SYMDEL statement.

ERROR: The variable value was previously declared as READONLY and cannot be re-declared.

Cause	Solution
An attempt was made to redefine a macro variable that was initialized with the READONLY option on a %GLOBAL or %LOCAL statement for a second time.	A macro variable can be defined only once on a %GLOBAL or %LOCAL statement when using the READONLY option.

SAS Macro Warning Messages

This section contains warning messages that might be reported when using macros and the solutions to correct them . If you are unable to resolve the warning, then contact SAS Technical Support.

`WARNING: Apparent symbolic reference value not resolved.`

Cause	Solution
A macro variable is being referenced but cannot be found.	Define the macro variable before resolution.
A macro variable was spelled incorrectly.	Verify the spelling of the macro variable.

A local macro variable to a specific macro is being used globally outside the macro.	Add the macro variable to a %GLOBAL statement, or if you are using CALL SYMPUT, then use CALL SYMPUTX with the third argument as `'g'`, for example,
	`call symputx('`*`macro_variable`*`',`
	`symbolic_reference_value``,'g');`

A macro variable is being used in the same step as a CALL SYMPUT routine.	A step boundary such as a RUN statement must be reached before resolving the macro variable created with CALL SYMPUT.

The macro resolution occurred within a macro when the macro variable was created with a CALL SYMPUT routine or the INTO clause. The macro is being invoked with a CALL EXECUTE routine.	Place the %NRSTR function around the macro invocation, for example,
	`call execute('%nrstr(%`*`macro_name`*
	`('||`*`variable1`*`||'))');`
	This delays the resolution.

You have omitted the period delimiter when adding text to the end of the macro variable.	When text follows a macro variable a period is needed after the macro variable name, for example,
	`%let var=abc;`
	`%put &var.def;`
	This code resolves to **`abcdef`**.

`WARNING: Apparent invocation of macro value not resolved.`

Cause	Solution
You have misspelled the macro name.	Verify the spelling of the macro name.
The MAUTOSOURCE system option is turned off.	If invoking an autocall macro, then the MAUTOSOURCE system option must be turned on.
The MAUTOSOURCE system option is on, but you have specified an incorrect pathname in the SASAUTOS= system option.	The SASAUTOS= system option must contain the exact path for the location of the macro.
You are using the autocall facility but you do not have permission to the path on the SASAUTOS= system option.	Ensure that you have Read or Write access to the directory.
You are using the autocall facility but you have given different names for the macro name and the file name.	When using autocall macros, the macro name must match the file name exactly.
You are using the autocall facility but did not give the file a .sas extension.	When using autocall macros, the file that contains the macro must have the .sas extension.
You are using the autocall facility but the file name contains mixed case.	When using autocall macros, on UNIX the file name must be all lowercase letters.
The macro has not been compiled.	The definition of a macro must be compiled before the invocation of a macro.

`Warning: Extraneous text on %MEND statement ignored for macro definition value.`

Cause	Solution
The name in the %MEND statement does not match the name in the %MACRO statement.	The name in the %MEND statement and the %MACRO statement must match.
There is a missing semicolon in the %MEND statement.	The %MEND statement requires a semicolon to end the statement.

`Warning: Argument value to macro function value is out of range.`

Cause	Solution
The first **value** represents the position of the argument that is causing the problem. The second **value** represents the function that is being used. This argument is either less than or greater than the range allowed.	Ensure that the argument is within the range needed, for example, `%put %scan(a b c,0);` The value **0** is less than the range allowed. `%put %substr(abc,4,1);` The value **4** is greater than the length of the first argument.

`Warning: Missing %MEND statement for macro value.`

Cause	Solution
There is an unclosed comment that is causing the %MEND statement not to be seen.	In the comments, ensure that every `/*` has a matching `*/` and that every `%*` and `*` have a matching semicolon. If running interactively, you might need to restart your Compute Server session after fixing the problem or you might try running the following code: `;*%mend;*);*';*";**/;run;`
There is a missing semicolon prior to the %MEND statement.	Ensure that each statement prior to the %MEND statement that requires a semicolon has one.
There is an unmatched quotation mark prior to the %MEND statement.	Ensure that every double quotation mark and single quotation mark have a matching quotation mark.
There is no %MEND statement.	Every %MACRO statement requires a matching %MEND statement.

`Warning: Source level autocall is not found or cannot be opened. Autocall has been suspended and OPTION NOMAUTOSOURCE has been set. To use the autocall facility again, set OPTION MAUTOSOURCE.`

Cause	Solution
All library specifications on the SASAUTOS= system option are invalid or do not exist.	Ensure that the locations on the SASAUTOS= system option are valid and exist. Use the MAUTOSOURCE system option and the MRECALL system option.

`Warning: The argument to macro function %SYSGET is not defined as a system variable.`

Cause	Solution
The value used within the %SYSGET function is not recognized as a valid environment variable.	Check the spelling and ensure that the value is a valid environment variable on your operating system.
Quotation marks were used around the value being passed to the %SYSGET function.	Remove the quotation marks. They are not needed within a macro function.

`Warning: The RESOLVE function is disabled by the NOMACRO option.`

Cause	Solution
The NOMACRO option has been set at invocation time.	To use any part of the macro facility the MACRO option must be set at SAS invocation.

`Warning: The value. SASMACR catalog is opened for read only.`

Cause	Solution
`value` represents the libref that is associated with the SASMACR catalog. The SASMSTORE= system option is pointing to a libref where the SASMACR catalog has been set with read-only attributes.	The stored compiled macro catalog is initially opened for Read-Only access. When a session first attempts to execute a stored compiled macro, the library is opened with no lock. The library remains in that state until the session either ends or attempts to add or update a macro. Therefore, the warning above is no longer generated.

`Warning: Argument value to function value referenced by the %SYSFUNC or %QSYSFUNC macro function is out of range.`

Cause	Solution
The first `value` represents the position of the argument causing the problem. The second `value` represents the function that is used. This argument is either less than or greater than the range allowed.	Ensure that the argument is within the range needed, for example, `%put %sysfunc(scan(a b c,0));` The value `0` is less than the range allowed. `%put %sysfunc(substr(abc,4,1));` The value `4` is greater than the length of the first argument.

`Warning: Missing semicolon between %THEN clause and value has been assumed.`

Cause	Solution

The action following the %THEN statement is missing a semicolon.	Add a semicolon following the action for the %THEN statement, for example,
	`%let var=;`
	`%macro test;`
	`%if 1=1 %then &var    ← missing semicolon`
	`%mend test;`

Warning: Attempt to delete macro variable value failed. Variable not found.

Cause	Solution
The macro variable referenced in a %SYMDEL statement does not exist.	Add the NOWARN option to the %SYMDEL statement, for example, `%symdel aa / nowarn;`
The macro variable referenced in a %SYMDEL statement started with an ampersand.	The %SYMDEL statement requires the macro variable name without the ampersand.

Warning: Extraneous text on %SYMDEL statement ignored.

Cause	Solution
There is text following the forward slash in the %SYMDEL statement.	Only the NOWARN argument is valid following the forward slash.

Warning: Extraneous text in second argument to SYMDEL routine ignored.

Cause	Solution
There is text following the comma that does not equal the NOWARN argument.	Only the NOWARN argument is valid as the second argument following the comma.

Warning: The macro value was compiled with the SAS System value. The current version of the SAS System is value. The macro might not execute correctly. To avoid this message, recompile the macro with the value version of the SAS System.

Cause	Solution
A macro is being invoked on a release of SAS other than the	Move macro source code to new SAS release and compile.

Cause	Solution
release of SAS where it was compiled.	
The SASMACR catalog has been moved to a different operating system or release of SAS.	Stored compiled macros cannot be moved from one operating system to another or from one SAS release to another. The source code for the macros contained within the SASMACR catalog needs to be moved to the new location and compiled.

`Warning: The text expression value contains a recursive reference to the macro variable value. The macro variable will be assigned the null value.`

Cause	Solution
A macro variable is being set to itself where the original macro variable did not exist, for example, `%let a=&a;` If `&a` did not exist prior to this %LET statement, then the warning is generated.	Ensure that the macro variable exists before setting it back to itself, for example, `%global a;` `%let a=&a;`

`Warning: Extraneous text on %SYSMSTORECLEAR statement ignored.`

Cause	Solution
There is text following the %SYSMSTORECLEAR statement.	Remove the text, nothing should fall between the %SYSMSTORECLEAR statement and the semicolon.

`Warning: Extraneous argument text on %SYSMACDELETE call ignored: value.`

Cause	Solution
There is text following the forward slash that does not equal the NOWARN argument.	Only the NOWARN argument is valid following the forward slash.
The macro name in the %SYSMACDELETE statement begins with a percent sign.	The %SYSMACDELETE statement requires the macro name without the percent sign.

`Warning: The MCOVERAGE option was set, but no MCOVERAGELOC= was specified.`

`Warning: Generation of coverage data has been suspended and OPTION NOCOVERAGE has been set. Any fileref used in the MCOVERAGELOC option has been deassigned. To generate coverage data again, set OPTIONS MCOVERAGE and MCOVERAGELOC.`

Cause	Solution
The MCOVERAGE system option has been set but no location has been specified on the MCOVERAGELOC= system option.	A valid location must be placed on the MCOVERAGELOC= system option. The MCOVERAGE system option must be specified again.
The MCOVERAGELOC= system option is pointing to a location that does not exist.	Ensure that the path on the MCOVERAGELOC= system option exists and is valid. The MCOVERAGE system option must be specified again.

`Warning: Attempt to delete macro definition for value failed. Macro definition not found.`

Cause	Solution
The macro name specified in the %SYSMACDELETE statement does not exist.	Add the NOWARN argument to the %SYSMACDELETE statement, for example, `%sysmacdelete abc / nowarn;`

`Warning: The macro value was compiled with ENCODING=value, this session is running with ENCODING=value.`

Cause	Solution
The macro being invoked was compiled on a system with a different setting of the ENCODING= system option than the machine where the macro is being invoked.	Ensure that the ENCODING= system option is set to the same system as the system where the macro was compiled.

Appendix 3
SAS Tokens

SAS Tokens

When SAS processes a program, a component called the word scanner reads the program, character by character, and groups the characters into words. These words are referred to as tokens.

List of Tokens

SAS recognizes four general types of tokens:

Literal
: One or more characters enclosed in single or double quotation marks. Examples of literals include the following:

Table A11.1 *Literals*

'CARY'	"2008"
'Dr. Kemple-Long'	'<entry align="center">'

Name

One or more characters beginning with a letter or an underscore. Other characters can be letters, underscores, and digits.

Table A11.2 Names

data	_test	linesleft
f25	univariate	otherwise
year_2008	descending	

Number

A numeric value. Number tokens include the following:

- integers. Integers are numbers that do not contain a decimal point or an exponent. Examples of integers include 1, 72, and 5000. SAS date, time, and datetime constants such as '24AUG2008'D are integers, as are hexadecimal constants such as 0C4X.
- real (floating-point) numbers. Floating-point numbers contain a decimal point or an exponent. Examples include numbers such as 2.35, 5., 2.3E1, and 5.4E− 1.

Special character

Any character that is not a letter, number, or underscore. The following characters are some special characters:

= + − % & ; ( ) #

The maximum length of any type of token is 32,767 characters. A token ends when the tokenizer encounters one of the following situations:

- the beginning of a new token.
- a blank after a name or number token.
- in a literal token, a quotation mark of the same type that started the token. There is an exception. A quotation mark followed by a quotation mark of the same type is interpreted as a single quotation mark that becomes part of the literal token. For example, in `'Mary''s'`, the fourth quotation mark terminates the literal token. The second and third quotation marks are interpreted as a single character, which is included in the literal token.

Appendix 4

Syntax for Selected Functions Used with the %SYSFUNC and %QSYSFUNC Functions

Summary Descriptions and Syntax

This appendix provides summary descriptions and syntax for selected functions that can be used with the %SYSFUNC function.

Functions and Arguments for %SYSFUNC and %QSYSFUNC

The following table shows the syntax for selected functions that can be used with the %SYSFUNC function. The functions can be used with the %QYSFUNC function a well. This is not a complete list of the functions that can be used. For a list of functions that cannot be used, see Table 17.51 on page 359.

Table A12.1 Functions and Arguments for %SYSFUNC

Function	Description and Syntax
ATTRC	Returns the value of a character attribute for a SAS data set. `%SYSFUNC(ATTRC(`*`data-set-id,attr-name`*`))`
ATTRN	Returns the value of a numeric attribute for specified SAS data set. `%SYSFUNC(ATTRN(`*`data-set_id,attr-name`*`))`
CEXIST	Verifies the existence of a SAS catalog or SAS catalog entry. `%SYSFUNC(CEXIST`*`(entry <, U>)`*`)`
CLOSE	Closes a SAS data set. `%SYSFUNC(CLOSE(`*`data-set-id`*`))`
CUROBS	Returns the number of the current observation. `%SYSFUNC(CUROBS(`*`data-set-id`*`))`
DCLOSE	Closes a directory. `%SYSFUNC(DCLOSE(`*`directory-id`*`))`
DINFO	Returns specified information items for a directory. `%SYSFUNC(DINFO(`*`directory-id,info-items`*`))`
DNUM	Returns the number of members in a directory. `%SYSFUNC(DNUM(`*`directory-id`*`))`
DOPEN	Opens a directory. `%SYSFUNC(DOPEN(`*`fileref`*`))`
DOPTNAME	Returns a specified directory attribute. `%SYSFUNC(DOPTNAME(`*`directory-id,nval`*`))`
DOPTNUM	Returns the number of information items available for a directory. `%SYSFUNC(DOPTNUM(`*`directory-id`*`))`
DREAD	Returns the name of a directory member. `%SYSFUNC(DREAD(`*`directory-id,nval`*`))`
DROPNOTE	Deletes a note marker from a SAS data set or an external file. `%SYSFUNC(DROPNOTE(`*`data-set-id`* | *`file-id,note-id`*`))`
DSNAME	Returns the data set name associated with a data set identifier.

Function	Description and Syntax
	`%SYSFUNC(DSNAME(<data-set-id>))`
EXIST	Verifies the existence of a SAS library member. `%SYSFUNC(EXIST(member-name<,member-type>))`
FAPPEND	Appends a record to the end of an external file. `%SYSFUNC(FAPPEND(file-id<,cc>))`
FCLOSE	Closes an external file, directory, or directory member. `%SYSFUNC(FCLOSE(file-id))`
FCOL	Returns the current column position in the File Data Buffer (FDB). `%SYSFUNC(FCOL(file-id))`
FDELETE	Deletes an external file. `%SYSFUNC(FDELETE(fileref))`
FETCH	Reads the next nondeleted observation from a SAS data set into the Data Set Data Vector (DDV). `%SYSFUNC(FETCH(data-set-id<,NOSET>))`
FETCHOBS	Reads a specified observation from a SAS data set into the DDV. `%SYSFUNC(FETCHOBS(data-set-id,obs-number<,options>))`
FEXIST	Verifies the existence of an external file associated with a fileref. `%SYSFUNC(FEXIST(fileref))`
FGET	Copies data from the FDB. `%SYSFUNC(FGET(file-id,cval<,length>))`
FILEEXIST	Verifies the existence of an external file by its physical name. `%SYSFUNC(FILEEXIST(file-name))`
FILENAME	Assigns or deassigns a fileref for an external file, directory, or output device. `%SYSFUNC(FILENAME(fileref,file-name<,device<,host-options<,dir-ref>>>))`
FILEREF	SAS sessionVerifies that a fileref has been assigned in the Compute Server. `%SYSFUNC(FILEREF(fileref))`
FINFO	Returns a specified information item for a file. `%SYSFUNC(FINFO(file-id,info-item))`

Function	Description and Syntax
FNOTE	Identifies the last record that was read. `%SYSFUNC(FNOTE(file-id))`
FOPEN	Opens an external file. `%SYSFUNC(FOPEN(fileref<,open-mode<,record-length<,record-format>>>))`
FOPTNAME	Returns the name of an information item for an external file. `%SYSFUNC(FOPTNAME(file-id,nval))`
FOPTNUM	Returns the number of information items available for an external file. `%SYSFUNC(FOPTNUM(file-id))`
FPOINT	Positions the read pointer on the next record to be read. `%SYSFUNC(FPOINT(file-id,note-id))`
FPOS	Sets the position of the column pointer in the FDB. `%SYSFUNC(FPOS(file-id,nval))`
FPUT	Moves data to the FDB of an external file starting at the current column position. `%SYSFUNC(FPUT(file-id,cval))`
FREAD	Reads a record from an external file into the FDB. `%SYSFUNC(FREAD(file-id))`
FREWIND	Positions the file pointer at the first record. `%SYSFUNC(FREWIND(file-id))`
FRLEN	Returns the size of the last record read, or the current record size for a file opened for output. `%SYSFUNC(FRLEN(file-id))`
FSEP	Sets the token delimiters for the FGET function. `%SYSFUNC(FSEP(file-id,cval))`
FWRITE	Writes a record to an external file. `%SYSFUNC(FWRITE(file-id<,cc>))`
GETOPTION	Returns the value of a SAS system or graphics option. `%SYSFUNC(GETOPTION(option-name<,reporting-options<,...>>))`
GETVARC	Assigns the value of a SAS data set variable to a character DATA step or macro variable.

Function	Description and Syntax
	`%SYSFUNC(GETVARC(`*`data-set-id,var-num`*`))`
GETVARN	Assigns the value of a SAS data set variable to a numeric DATA step or macro variable. `%SYSFUNC(GETVARN(`*`data-set-id,var-num`*`))`
LIBNAME	Assigns or deassigns a libref for a SAS library. `%SYSFUNC(LIBNAME(`*`libref<,SAS-data-library<,engine<,options>>>`*`))`
LIBREF	Verifies that a libref has been assigned. `%SYSFUNC(LIBREF(`*`libref`*`))`
MOPEN	Opens a directory member file. `%SYSFUNC(MOPEN(`*`directory-id,member-name<open-mode<,record-length<,record-format>>>`*
NOTE	Returns an observation ID for current observation of a SAS data set. `%SYSFUNC(NOTE(`*`data-set-id`*`))`
OPEN	Opens a SAS data file. `%SYSFUNC(OPEN(`*`<data-file-name<,mode>>`*`))`
PATHNAME	Returns the physical name of a SAS library or an external file. `%SYSFUNC(PATHNAME(`*`fileref`*`))`
POINT	Locates an observation identified by the NOTE function. `%SYSFUNC(POINT(`*`data-set-id,note-id`*`))`
REWIND	Positions the data set pointer to the beginning of a SAS data set. `%SYSFUNC(REWIND(`*`data-set-id`*`))`
SPEDIS	Returns a number for the operation required to change an incorrect keyword in a WHERE clause to a correct keyword. `%SYSFUNC(SPEDIS(`*`query,keyword`*`))`
SYSGET	Returns the value of the specified host environment variable. `%SYSFUNC(sysget(`*`host-variable`*`))`
SYSRC	Returns the system error number or exit status of the entry most recently called. `%SYSFUNC(SYSRC())`
VARFMT	Returns the format assigned to a data set variable. `%SYSFUNC(VARFMT(`*`data-set-id,var-num`*`))`

Function	Description and Syntax
VARINFMT	Returns the informat assigned to a data set variable. `%SYSFUNC(VARINFMT(`*`data-set-id`*`,`*`var-num`*`))`
VARLABEL	Returns the label assigned to a data set variable. `%SYSFUNC(VARLABEL(`*`data-set-id`*`,`*`var-num`*`))`
VARLEN	Returns the length of a data set variable. `%SYSFUNC(VARLEN(`*`data-set-id`*`,`*`var-num`*`))`
VARNAME	Returns the name of a data set variable. `%SYSFUNC(VARNAME(`*`data-set-id`*`,`*`var-num`*`))`
VARNUM	Returns the number of a data set variable. `%SYSFUNC(VARNUM(`*`data-set-id`*`,`*`var-name`*`))`
VARTYPE	Returns the data type of a data set variable. `%SYSFUNC(VARTYPE(`*`data-set-id`*`,`*`var-num`*`))`

Appendix 5

SAS Macro Examples

Example 1: Import All CSV Files That Exist within a Directory

Details

This example uses the IMPORT procedure to import each CSV file that resides within a directory that is passed to the macro.

Program

```
%macro drive(dir,ext);
   %local cnt filrf rc did memcnt name;

   %let cnt=0;
   %let filrf=mydir;

   %let rc=%sysfunc(filename(filrf,&dir));
   %let did=%sysfunc(dopen(&filrf));

   %if &did ne 0 %then %do;
      %let memcnt=%sysfunc(dnum(&did));
      %do i=1 %to &memcnt;
         %let name=%qscan(%qsysfunc(dread(&did,&i)),-1,.);
         %if %qupcase(%qsysfunc(dread(&did,&i))) ne %qupcase(&name)
%then %do;
            %if %superq(ext) = %superq(name) %then %do;
               %let cnt=%eval(&cnt+1);
               %put %qsysfunc(dread(&did,&i));
               proc import datafile="&dir\%qsysfunc(dread(&did,&i))"
out=dsn&cnt
                  dbms=csv replace;
                  guessingrows=max;
               run;
            %end;
         %end;
       %end;
    %end;
  %else %put &dir cannot be opened.;

  %let rc=%sysfunc(dclose(&did));
%mend drive;
```

```
%drive(c:\temp,csv)
```

Program Description

Begin the macro definition with two parameters.

```
%macro drive(dir,ext);
   %local cnt filrf rc did memcnt name;
   %let cnt=0;
```

Create a macro variable named &FILRF and assign it the value Mydir. Mydir will be used as the name of a fileref to the directory stored in macro variable DIR.

```
   %let filrf=mydir;
```

Create a macro variable named RC that uses the FILENAME function to create a fileref to the path stored in macro variable DIR. FILRF is a reference to macro variable FILRF, which contains MYDIR, without the ampersand. When function FILENAME is invoked with %SYSFUNC, the *fileref* argument must be the name of a macro variable without the ampersand. See *fileref* in "FILENAME Function" in *SAS Functions and CALL Routines: Reference*. The FILENAME function result is stored in macro variable RC.

```
   %let rc=%sysfunc(filename(filrf,&dir));
```

Create a macro variable named DID that uses the DOPEN function to open the directory. Macro variable FILRF contains MYDIR, the name of the fileref to the directory stored in macro variable DIR. The DOPEN function returns a directory identifier value. The returned value is `0` if the directory cannot be opened.

```
   %let did=%sysfunc(dopen(&filrf));
```

Use a %IF condition to make sure the directory was opened successfully.

```
   %if &did ne 0 %then %do;
```

Create a macro variable named MEMCNT that uses the DNUM function to return the number of members within the directory.

```
      %let memcnt=%sysfunc(dnum(&did));
```

Use the %DO statement to loop through the entire directory based on the number of members returned from the DNUM function.

```
      %do i=1 %to &memcnt;
```

Create a macro variable named NAME that uses the DREAD function to return the name of each file. The %QSCAN function uses `-1` as the second argument to start at the end of the file name. The third argument uses the period as the delimiter. This causes the extension of the file to be returned and to be assigned to the macro variable NAME.

```
         %let name=%qscan(%qsysfunc(dread(&did,&i)),-1,.);
```

Use the %IF statement to verify that each file contains an extension. If the file does not contain an extension, then the contents of the %DO statement do not execute.

```
         %if %qupcase(%qsysfunc(dread(&did,&i))) ne %qupcase(&name) %then %do;
```

Use the %IF statement to verify that the extension matches the second parameter that is passed to the macro. If the condition is true, then increment a counter (&CNT) by `1`. The %PUT statement prints the full name to the log.

The IMPORT procedure is used to read each .csv file that is returned and creates a data set named DSN**x**, where **x** is from the counter macro variable named CNT. By default, the IMPORT procedure scans the first 20 rows of the input data to determine the variable types and lengths in the imported data set. For character data, if values of a longer length exist in subsequent rows in the input data, those values might be truncated in the imported data set. To avoid that issue, the GUESSINGROWS=MAX statement is specified in the IMPORT procedure step to force the IMPORT procedure to scan all of the input data rows instead of the first 20. See “GUESSINGROWS Statement” in *Base SAS Procedures Guide*. If this is not a potential issue with your input data, you can eliminate the GUESSINGROWS statement.

```
            %if %superq(ext) = %superq(name) %then %do;
               %let cnt=%eval(&cnt+1);
               %put %qsysfunc(dread(&did,&i));
               proc import datafile="&dir\%qsysfunc(dread(&did,&i))" out=dsn&cnt
                  dbms=csv replace;
                  guessingrows=max;
               run;
            %end;
```

Use the %END statements to close the %DO blocks.

```
         %end;
       %end;
    %end;
```

If the directory cannot be opened, the %ELSE is executed to write a note to the log:

```
  %else %put &dir cannot be open.;
```

Use the DCLOSE function to close the directory.

```
  %let rc=%sysfunc(dclose(&did));
```

Invoke the macro. The first parameter is the directory where the files exist. The second parameter is the extension of the files that you are importing.

```
%drive(c:\temp,csv)
```

Example 2: List All Files within a Directory Including Subdirectories

Details

This macro lists all files within the directory that is passed to the macro including any subdirectories.

Note: This example will not run if you use the LOCKDOWN system option.

Program

```
%macro list_files(dir,ext);
  %local filrf rc did memcnt name i;
  %let rc=%sysfunc(filename(filrf,&dir));
  %let did=%sysfunc(dopen(&filrf));

   %if &did eq 0 %then %do;
    %put Directory &dir cannot be open or does not exist;
    %return;
  %end;

   %do i = 1 %to %sysfunc(dnum(&did));

   %let name=%qsysfunc(dread(&did,&i));

      %if %qupcase(%qscan(&name,-1,.)) = %upcase(&ext) %then %do;
        %put &dir\&name;
      %end;
      %else %if %qscan(&name,2,.) = %then %do;
        %list_files(&dir\&name,&ext)
      %end;

   %end;
   %let rc=%sysfunc(dclose(&did));
   %let rc=%sysfunc(filename(filrf));

%mend list_files;
%list_files(c:\temp,sas)
```

Program Description

Begin the macro definition with two parameters.

```
%macro list_files(dir,ext);
  %local filrf rc did memcnt name i;
```

Create two macro variables. Macro variable RC will contain the results from the FILENAME function. The value is `0` if it is successful or nonzero if it is not successful. The FILENAME function assigns to macro variable FILRF a system-generated fileref to the directory that is passed to the macro (&DIR). Macro variable DID contains the results from the DOPEN function that opens the directory. DOPEN returns a directory identifier value of `0` if the directory cannot be opened.

```
  %let rc=%sysfunc(filename(filrf,&dir));
  %let did=%sysfunc(dopen(&filrf));
```

Use the %IF statement to make sure the directory can be opened. If not, end the macro.

```
   %if &did eq 0 %then %do;
    %put Directory &dir cannot be open or does not exist;
    %return;
  %end;
```

Use the %DO statement to loop through the entire directory based on the number of members returned from the DNUM function.

```
   %do i = 1 %to %sysfunc(dnum(&did));
```

Create a macro variable named NAME that will contain each file name in the directory that is passed to the macro. The DREAD function is used to retrieve the name of each file.

```
   %let name=%qsysfunc(dread(&did,&i));
```

Use the %IF statement to see whether the extension matches the second parameter value that is supplied to the macro. If the condition is true, then print the full name to the log. %QSCAN is used to pull off the extension of the file name (&NAME) by using -1 as the second argument. %QUPCASE is used on both sides of the equal to ensure the case matches.

```
      %if %qupcase(%qscan(&name,-1,.)) = %upcase(&ext) %then %do;
        %put &dir\&name;
      %end;
```

If the previous %IF condition is false, then use %ELSE %IF statements to see whether the name contains an extension. If no extension is found, then assume that the name is a directory and call the macro again. This ensures that all subdirectories are read.

```
      %else %if %qscan(&name,2,.) = %then %do;
        %list_files(&dir\&name,&ext)
      %end;
```

Use the %END statement to close the %DO block.

```
   %end;
```

Use the DCLOSE function to close the directory. Use the FILENAME function to unassign the fileref.

```
   %let rc=%sysfunc(dclose(&did));
   %let rc=%sysfunc(filename(filrf));

%mend list_files;
```

Invoke the macro. The first parameter is the directory where the files exist. The second parameter is the extension of the files that you want to list.

```
  %list_files(/home/suholm/sas)
```

Example 3: How to Increment a Macro DO Loop by a Non-integer Value

Details

The iterative %DO loop within the macro facility can handle only integer values or expressions that generate integer values. This goes for the %BY portion also. This macro illustrates how to loop through values using a non-integer %BY value. Some arithmetic and various macro functions are used to accomplish the task.

Program

```
%macro loop( start= , end= , by= ) ;
%local i;
%do i = 1 %to %eval(%sysfunc(Ceil(%sysevalf((&end - &start ) /
&by ) ) ) +1) ;
   %let value=%sysevalf(( &start - &by ) + ( &by * &i )) ;
   %if &value <=&end %then %do;
      %put &value;
   %end;
%end ;
%mend loop ;
%loop(start = 1 , end = 5 , by = .25 )
```

Program Description

Begin the macro definition with three keyword parameters.

```
%macro loop( start= , end= , by= ) ;
%local i;
```

When using nested functions, you start inside and work your way out. The inner most %SYSEVALF function is used to subtract &END by &START and then divide that result by the value of &BY. This method returns a whole number. The CEIL function is used to return the smallest integer that is greater than or equal to the argument. To complete the formula the %EVAL function is used to add 1 to the final value. Use %DO to loop through the number returned from this formula.

```
%do i = 1 %to %eval(%sysfunc(Ceil(%sysevalf((&end - &start ) / &by ) ) ) +1) ;
```

Create a macro variable named VALUE that uses the %SYSEVALF function to re-create the value as a non-integer. First subtract &START by &BY and then add that to the product of &BY and &I.

```
   %let value=%sysevalf(( &start - &by ) + ( &by * &i )) ;
```

Use the %IF statement to check that &VALUE from the %LET statement is less than or equal to &END. If the condition is true, then place the value in the log using the %PUT statement.

```
%if &value <=&end %then %do;
      %put &value;
```

Use the %END statement to close both %DO loop blocks.

```
   %end;
%end ;
%mend loop ;
```

Invoke the macro by using the start and end values of the loop as well as the increment value.

```
%loop(start = 1 , end = 5 , by = .25 )
```

Example 4: How to Use Character Values on a Macro %DO Loop

Details

The macro facility does not allow character values on an iterative %DO loop. There are two macros below that give a work around to that limitation. These two macros use different techniques to step through single-byte characters that are passed to the macro.

Program

Example 4A: This example enables you to step through only the characters that are passed to the macro as a parameter to the macro variable named LST.

```
%macro iterm(lst);
 %let finish=%sysfunc(countw(&lst));
  %do i = 1 %to &finish;
   %put %scan(&lst,&i);
  %end;
%mend iterm;

%iterm(a c e)
```

Example 4B: This example enables you to step through the characters between the &BEG and &END values.

```
%macro iterm(beg,end);
%do i = %sysfunc(rank(&beg)) %to %sysfunc(rank(&end));
  %put %sysfunc(byte(&i));
%end;
%mend iterm;

%iterm(a,e)
```

Program Description

Example 4A

Begin the macro definition with one parameter.

```
%macro iterm(lst);
```

Create the macro variable named FINISH to contain the number of characters or words in the macro variable named LST.

```
 %let finish=%sysfunc(countw(&lst));
```

Use the %DO statement to loop through the number of characters or words contained within &LST.

```
%do i = 1 %to &finish;
```

Use the %SCAN function to pull off each character or word in the macro variable named LST and use the %PUT statement to write contents to the log.

```
   %put %scan(&lst,&i);
```

End the %DO block.

```
  %end;
```

End the macro.

```
%mend iterm;
```

Invoke the macro that passes in the characters to loop through, which are separated by a space.

```
%iterm(a c e)
```

Here are the results for example, 4A.

Example 4B

Begin the macro definition with two parameters.

```
%macro iterm(beg,end);
```

Use the RANK function to return the position of a character in the ASCII or EBCDIC collating sequence. The %SYSFUNC function is needed to use SAS functions within the macro facility. Use the %DO statement to loop through the number of words between &BEG and &END.

```
%do i = %sysfunc(rank(&beg)) %to %sysfunc(rank(&end));
```

Use the BYTE function to return the character in the ASCII or the EBCDIC collating sequence that is represented from the macro variable named I. The %PUT statement is used to write the contents to the log.

```
  %put %sysfunc(byte(&i));
```

End the %DO block.

```
%end;
```

End the macro.

```
%mend iterm;
```

Invoke the macro that uses the starting character and ending character.

```
%iterm(a,e)
```

Here are the results for example, 4B.

Logs

Log: Example 4A

```
1  %macro iterm(lst);
2   %let finish=%sysfunc(countw(&lst));
3    %do i = 1 %to &finish;
4     %put %scan(&lst,&i);
5    %end;
6  %mend iterm;
7
8  %iterm(a c e)
a
c
e
```

Log: Example 4B

```
180  %macro iterm(beg,end);
181  %do i = %sysfunc(rank(&beg)) %to %sysfunc(rank(&end));
182    %put %sysfunc(byte(&i));
183  %end;
184  %mend iterm;
185  /* Just pass in starting and ending value */
186  %iterm(a,e)
a
b
c
d
e
```

Example 5: Place All SAS Data Set Variables into a Macro Variable

Details

This macro uses the ATTRN and VARNAME functions to retrieve all the variables contained within a SAS data set and places them into one macro variable.

Program

```
data one;
  input x y;
datalines;
1 2
;

%macro lst(dsn);
  %local dsid cnt rc;
  %global x;
  %let x=;
  %let dsid=%sysfunc(open(&dsn));
   %if &dsid ne 0 %then %do;  %let cnt=%sysfunc(attrn(&dsid,nvars));

   %do i = 1 %to &cnt;
    %let x=&x %sysfunc(varname(&dsid,&i));
   %end;

  %end;
    %else %put &dsn cannot be open.;
  %let rc=%sysfunc(close(&dsid));

%mend lst;

%lst(one)

%put macro variable x = &x;
```

Program Description

Create a data set.

```
data one;
  input x y;
datalines;
1 2
;
```

Begin the macro definition that contains one parameter.

```
%macro lst(dsn);
```

Create the macro variable named DSID that uses the OPEN function to open the data set that is passed to the macro. Use the %IF condition to make sure the data set opened successfully. If the value is yes, then create the macro variable named CNT that uses the ATTRN function with the NVARS argument to return the number of variables that are contained within the SAS data set.

```
%local dsid cnt rc;
  %global x;
  %let x=;
  %let dsid=%sysfunc(open(&dsn));
    %if &dsid ne 0 %then %do;  %let cnt=%sysfunc(attrn(&dsid,nvars));
Use the %DO statement to loop through the number of variables within the SAS data set.
     %do i = 1 %to &cnt;
```

Create the macro variable named X that uses the VARNAME function to return the name of a SAS data set variable. The first argument to this function specifies the SAS data set identifier from the OPEN function. The second argument is the number of the variable's position. Each time through the %DO loop the value of &X is appended to itself.

```
    %let x=&x %sysfunc(varname(&dsid,&i));
```

End the %DO blocks.

```
 %end;
%end;
```

If the data set could not be opened, %ELSE is executed, which uses %PUT to write a note to the log.

```
   %else %put &dsn cannot be open.;
```

Use the CLOSE function to close the SAS data set.

```
  %let rc=%sysfunc(close(&dsid));
```

End the macro.

```
%mend lst;
```

Invoke the macro using the name of the SAS data set.

```
%lst(one)
```

Use the %PUT statement to write the contents of macro variable X to the log.

```
%put macro variable x = &x;
```

Log

```
1     %let cnt=%sysfunc(attrn(&dsid,nvars));
2
3     /* Create a macro variable that contains all dataset variables */
4      %do i = 1 %to &cnt;
5       %let x=&x %sysfunc(varname(&dsid,&i));
6      %end;
7
8     /* Close the data set */
9     %let rc=%sysfunc(close(&dsid));
10
11   %mend lst;
12
13     /* Pass in the name of the data set */
14   %lst(one)
15
16   %put macro variable x = &x;
macro variable x = x y
```

Example 6: Using a Macro to Create New Variable Names from Variable Values

Details

This example shows how to create new variable names using the values of existing variables. Macro processing is required to store the values within macro variables, and then assign each new variable the desired value in a subsequent DATA step. The desired result is to create new variables using each team color concatenated with the team name and the word Total. And output a single observation assigning each new variable the coordinating total sum of the three game scores.

Program

```
data teams;
  input color $15. @16 team_name $15. @32 game1 game2 game3;
datalines;
Green          Crickets        10 7 8
Blue           Sea Otters      10 6 7
Yellow         Stingers        9 10 9
Red            Hot Ants        8 9 9
Purple         Cats            9 9 9
;

%macro newvars(dsn);

data _null_;
  set &dsn end=end;
  count+1;
  call symputx('macvar'||left(count),compress(color)||
compress(team_name)||"Total");
  if end then call symputx('max',count);
run;

data teamscores;
  set &dsn end=end;

%do i = 1 %to &max;
  if _n_=&i then do;
    &&macvar&i=sum(of game1-game3);
    retain &&macvar&i;
    keep &&macvar&i;
  end;
%end;
if end then output;

%mend newvars;

options mprint;
%newvars(teams)
options nomprint;

proc print noobs;
  title "League Team Game Totals";
run;
```

Program Description

Create the data set.

```
data teams;
  input color $15. @16 team_name $15. @32 game1 game2 game3;
datalines;
Green          Crickets        10 7 8
Blue           Sea Otters      10 6 7
Yellow         Stingers        9 10 9
Red            Hot Ants        8 9 9
Purple         Cats            9 9 9
;
```

Begin the macro definition with one parameter.

```
  %macro newvars(dsn);
```

Use the DATA _NULL_ step along with CALL SYMPUTX call routine to create a separate macro variable for each observation read from the Teams data set. The END= option is used in the SET statement to create a variable that indicates when the end of file is reached. A counter is created in a variable named COUNT to count the total number of observations read. In the CALL SYMPUTX call routine, the name of each macro variable will have the COUNT value appended to the end of name. This routine creates macro variables named MACVAR1, MACVAR2, and so on. The values passed to each macro variable is the concatenation of the COLOR, TEAM_NAME, and the word Total. The COMPRESS function is used to remove any blank spaces in the COLOR or TEAM_NAME value since SAS variable names cannot contain blanks. The last IF condition creates a single macro variable named MAX that contains the final COUNT value.

```
data _null_;
  set &dsn end=end;
  count+1;
  call symputx('macvar'||left(count),compress(color)||compress(team_name)||"Total");
  if end then call symputx('max',count);
run;
```

Create a new data set named TeamScores and use the SET statement to bring in the data set that is passed to the macro.

```
data teamscores;
  set &dsn end=end;
```

Use the %DO statement to loop through the total number of observations (&MAX).

```
%do i = 1 %to &max;
```

When the %DO loop index variable &I matches the DATA step iteration variable _N_ the new variable for the current COLOR and TEAM_NAME is assigned the sum of the GAME1-GAME3 variables using the SUM function. This is done by using macro indirect referencing. When you use an indirect macro variable reference, you force the macro processor to scan the macro variable reference more than once and resolve the desired reference on the second, or later, scan. To force the macro processor to rescan a macro variable reference, you use more than one ampersand in the macro variable reference. When the macro processor encounters multiple ampersands, its basic action is to resolve two ampersands to one ampersand. The RETAIN statement is needed to retain the values of the new variables throughout the DATA step in order to output them all on a single observation. The KEEP statement is used to output only the new variables to the new data set. The %END statement closes the %DO block. The final IF statement is used to check for the ending of the data set and outputs the single observation.

```
  if _n_=&i then do;
    &&macvar&i=sum(of game1-game3);
    retain &&macvar&i;
    keep &&macvar&i;
  end;
%end;
if end then output;
```

End the macro.

```
%mend newvars;
```

Enable system option MPRINT. This is not required, but it can be helpful when debugging your macro code.

```
options mprint;
```

Invoke the macro using the name of the data set containing the variables to change,and then disable system option mprint.

```
%newvars(teams)
options nomprint;
```

Use PROC PRINT to print the results of the data set TeamScores.

```
proc print noobs;
  title "League Team Game Totals";
run;
```

Log

The following MPRINT messages are written to the SAS log:

```
...
MPRINT(NEWVARS):   data _null_;
MPRINT(NEWVARS):   set teams end=end;
MPRINT(NEWVARS):   count+1;
MPRINT(NEWVARS):   call
symputx('macvar'||left(count),compress(color)||compress(team_name)||"Total");
MPRINT(NEWVARS):   if end then call symputx('max',count);
MPRINT(NEWVARS):   run;

...

MPRINT(NEWVARS):   data teamscores;
MPRINT(NEWVARS):   set teams end=end;
MPRINT(NEWVARS):   if _n_=1 then do;
MPRINT(NEWVARS):   GreenCricketsTotal=sum(of game1-game3);
MPRINT(NEWVARS):   retain GreenCricketsTotal;
MPRINT(NEWVARS):   keep GreenCricketsTotal;
MPRINT(NEWVARS):   end;
MPRINT(NEWVARS):   if _n_=2 then do;
MPRINT(NEWVARS):   BlueSeaOttersTotal=sum(of game1-game3);
MPRINT(NEWVARS):   retain BlueSeaOttersTotal;
MPRINT(NEWVARS):   keep BlueSeaOttersTotal;
MPRINT(NEWVARS):   end;
MPRINT(NEWVARS):   if _n_=3 then do;
MPRINT(NEWVARS):   YellowStingersTotal=sum(of game1-game3);
MPRINT(NEWVARS):   retain YellowStingersTotal;
MPRINT(NEWVARS):   keep YellowStingersTotal;
MPRINT(NEWVARS):   end;
MPRINT(NEWVARS):   if _n_=4 then do;
MPRINT(NEWVARS):   RedHotAntsTotal=sum(of game1-game3);
MPRINT(NEWVARS):   retain RedHotAntsTotal;
MPRINT(NEWVARS):   keep RedHotAntsTotal;
MPRINT(NEWVARS):   end;
MPRINT(NEWVARS):   if _n_=5 then do;
MPRINT(NEWVARS):   PurpleCatsTotal=sum(of game1-game3);
MPRINT(NEWVARS):   retain PurpleCatsTotal;
MPRINT(NEWVARS):   keep PurpleCatsTotal;
MPRINT(NEWVARS):   end;
MPRINT(NEWVARS):   if end then output;
```

Output

Output A13.1 *PRINT Procedure Output*

League Team Game Totals

GreenCricketsTotal	BlueSeaOttersTotal	YellowStingersTotal	RedHotAntsTotal	PurpleCatsTotal
25	23	28	26	27

Example 7: Dynamically Determine the Number of Observations and Variables in a SAS Data Set

Details

This macro uses the functions OPEN and ATTRN to retrieve the total number of observations and variables contained within a SAS data set.

Program

```
data test;
   input a b c $ d $;
   datalines;
1 2 A B
3 4 C D
;

%macro obsnvars(ds);
   %global dset nvars nobs;
   %let dset=&ds;
   %let dsid = %sysfunc(open(&dset));

   %if &dsid %then %do;
      %let nobs =%sysfunc(attrn(&dsid,nlobs));
      %let nvars=%sysfunc(attrn(&dsid,nvars));
      %let rc = %sysfunc(close(&dsid));
   %end;

   %else %put open for data set &dset failed;
%mend obsnvars;

%obsnvars(test)

%put &dset has &nvars variable(s) and &nobs observation(s).;
```

Program Description

Create a data set named Test.

```
data test;
   input a b c $ d $;
   datalines;
1 2 A B
3 4 C D
;
```

Begin the macro definition with one parameter.

```
%macro obsnvars(ds);
```

Create a global macro variable DSET from the macro parameter DS by using %GLOBAL and %LET statements.

```
  %global dset nvars nobs;
   %let dset=&ds;
```

Use the OPEN function to open the data set that is passed to the macro.

```
   %let dsid = %sysfunc(open(&dset));
```

Use the %IF statement to make sure the data set was open. If it is open, then run three %LET statements.

- The first %LET statement creates a macro variable named NOBS. The ATTRN function is used with the NLOBS argument to retrieve the logical number of observations in the data set.
- The second %LET statement creates a macro variable named NVARS. The ATTRN function is used along with the NVARS argument to retrieve the number of variables in the data set.
- The third %LET statement uses the CLOSE function to close the data set that is passed to the macro.

Macro variable DSID contains the identifier that was returned from the OPEN function.

```
   %if &dsid %then %do;
      %let nobs =%sysfunc(attrn(&dsid,nlobs));
      %let nvars=%sysfunc(attrn(&dsid,nvars));
      %let rc = %sysfunc(close(&dsid));
   %end;
```

If the %IF condition is false, meaning that the data set is not open, then use the %PUT statement to write a statement to the log.

```
   %else %put open for data set &dset failed;
```

Close the macro definition.

```
%mend obsnvars;
```

Invoke the macro by passing in the data set name as a parameter.

```
%obsnvars(test)
```

Use the %PUT statement to write the information returned from the macro to the log.

```
%put &dset has &nvars variable(s) and &nobs observation(s).;
```

Log

```
2   data test;
83      input a b c $ d $;
84      datalines;
NOTE: The data set WORK.TEST has 2 observations and 4 variables.
NOTE: DATA statement used (Total process time):
      real time           0.00 seconds
      cpu time            0.01 seconds

87   ;
88
89   %macro obsnvars(ds);
90      %global dset nvars nobs;
91      %let dset=&ds
92      %let dsid = %sysfunc(open(&dset));
93
94      %if &dsid %then %do;
95         %let nobs =%sysfunc(attrn(&dsid,nlobs));
96         %let nvars=%sysfunc(attrn(&dsid,nvars));
97         %let rc = %sysfunc(close(&dsid));
98      %end;
99
100     %else %put open for data set dset failed;
101  %mend obsnvars;
102
103  %obsnvars(test)
104
105  %put &dset has &nvars variable(s) and &nobs observation(s).;
test has 4 variable(s) and 2 observation(s).
106
```

Example 8: Use Macro Logic to Determine If an External File Is Empty

Details

This macro verifies that an external file exists. If it does not exist, then a message is written to the log. If the file does exist, then the file is opened and SAS attempts to read the data within the file. Functions, such as FOPEN, FREAD and FGET, are used to retrieve the data. If there is no data to be read, a message is written to the log that the file is empty.

Program

```
%macro test(outf);
 %let filrf=myfile;
 %if %sysfunc(fileexist(&outf)) %then %do;
 %let rc=%sysfunc(filename(filrf,&outf));
 %let fid=%sysfunc(fopen(&filrf));
  %if &fid > 0 %then %do;
   %let rc=%sysfunc(fread(&fid));
   %let rc=%sysfunc(fget(&fid,mystring));
    %if &rc = 0 %then %put &mystring;
    %else %put file is empty;
   %let rc=%sysfunc(fclose(&fid));
  %end;
%let rc=%sysfunc(filename(filrf));
%end;
%else %put file does not exist;
%mend test;

%test(c:\test.txt)
```

Program Description

Begin the macro definition with one parameter.

```
%macro test(outf);
```

Use the %LET statement to create a macro variable named FILRF that will contain the fileref name that will be used later.

```
 %let filrf=myfile;
```

Use the %IF statement with the FILEEXIST function to verify that the file that is passed to the macro does exist.

```
 %if %sysfunc(fileexist(&outf)) %then %do;
```

Use the FILENAME function to associate a fileref with the file that is passed into the macro named OUTF. Use the FOPEN function to open that file and assign a file identifier value to the macro variable FID.

```
 %let rc=%sysfunc(filename(filrf,&outf));
 %let fid=%sysfunc(fopen(&filrf));
```

Use the %IF statement to make sure the file was successfully opened. FOPEN returns a `0` if the file could not be open and a nonzero if the file was successfully open.

```
  %if &fid > 0 %then %do;
```

Use the FREAD function to read a record from the file. Use the FGET function to copy the data into a macro variable named MYSTRING.

```
     %let rc=%sysfunc(fread(&fid));
     %let rc=%sysfunc(fget(&fid,mystring));
```

Use the %IF statement to make sure the FGET function was successful. The FGET function returns a `0` if the operation was successful. If the FGET function was successful, then use a %PUT statement to write the contents of the variable named MYSTRING to the log.

```
      %if &rc = 0 %then %put &mystring;
```

If the %IF condition is false, then use a %PUT statement to write a message to the log stating that the file is empty.

```
      %else %put file is empty;
```

Use the FCLOSE function to close the file.

```
     %let rc=%sysfunc(fclose(&fid));
```

End the inner %IF %THEN %DO block.

```
    %end;
```

Use the FILENAME function to disassociate the fileref from the file.

```
  %let rc=%sysfunc(filename(filrf));
```

End the outer %IF %THEN %DO block.

```
  %end;
```

If the %IF statement that uses the FILEEXIST function is false, then use the %PUT statement to write that the file does not exist to the log.

```
  %else %put file does not exist;
```

End the macro.

```
  %mend test;
```

Invoke the macro by using the name of the file to check for data.

```
  %test(c:\test.txt)
```

Example 9: Retrieve Each Word from a Macro Variable List

Details

This macro retrieves each word from a list, adds an underscore to the beginning of each word, and then renames variables based on these new values.

Program

```
%let varlist = Age Height Name Sex Weight;

%macro rename;
   %let word_cnt=%sysfunc(countw(&varlist));
   %do i = 1 %to &word_cnt;
     %let temp=%qscan(%bquote(&varlist),&i);
     &temp = _&temp
   %end;
%mend rename;

data new;
   set sashelp.class(rename=(%unquote(%rename)));
run;

proc print;
run;
```

Program Description

Use the %LET statement to create a macro variable named VARLIST that contains a list of words. In this case the words are a list of variables that will be renamed.

```
%let varlist = Age Height Name Sex Weight;
```

Begin the macro definition.

```
%macro rename;
```

Use the %LET statement to create a macro variable named WORD_CNT. Use the COUNTW function to receive the number of words in the list.

```
 %let word_cnt=%sysfunc(countw(&varlist));
```

Use a %DO statement to loop through the number of words specified in macro variable WORD_CNT. Use the %LET statement to create a macro variable named TEMP that uses the %QSCAN function to mask and retrieve each word from the macro variable VARLIST. The %BQUOTE function is used to mask any special characters that are in macro variable VARLIST.

```
   %do i = 1 %to &word_cnt;
     %let temp=%qscan(%bquote(&varlist),&i);
```

Generate the code that will be returned to the RENAME option. It adds an underscore to each word creating syntax such as `age=_age`.

```
     &temp = _&temp
```

End the %DO block.

```
   %end;
```

End the macro.

```
%mend rename;
```

Use the RENAME = option in the SET statement to invoke the macro that renames each variable in the macro variable named VARLIST. The %UNQUOTE function is used to remove the masking placed on the values using the %QSCAN function.

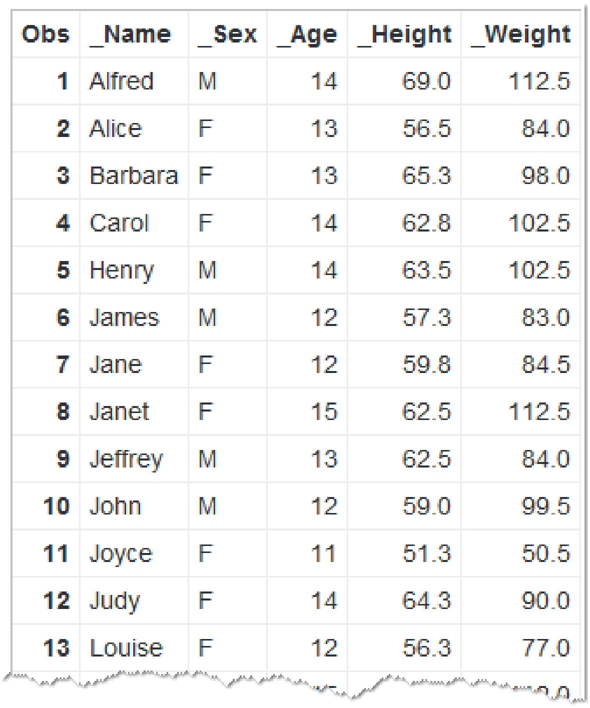
```
data new;
   set sashelp.class(rename=(%unquote(%rename)));
run;
```

Use PROC PRINT to display the results.

```
proc print;
run;
```

Output

Output A13.2 Partial PRINT Procedure Output



Obs	_Name	_Sex	_Age	_Height	_Weight
1	Alfred	M	14	69.0	112.5
2	Alice	F	13	56.5	84.0
3	Barbara	F	13	65.3	98.0
4	Carol	F	14	62.8	102.5
5	Henry	M	14	63.5	102.5
6	James	M	12	57.3	83.0
7	Jane	F	12	59.8	84.5
8	Janet	F	15	62.5	112.5
9	Jeffrey	M	13	62.5	84.0
10	John	M	12	59.0	99.5
11	Joyce	F	11	51.3	50.5
12	Judy	F	14	64.3	90.0
13	Louise	F	12	56.3	77.0

Example 10: Loop through Dates Using a Macro %DO Loop

Details

This macro illustrates how to loop through a starting and ending date by month.

Program

```
%macro date_loop(start,end);
   %let start=%sysfunc(inputn(&start,anydtdte9.));
   %let end=%sysfunc(inputn(&end,anydtdte9.));
   %let dif=%sysfunc(intck(month,&start,&end));
      %do i=0 %to &dif;
         %let date=%sysfunc(intnx(month,&start,&i,b),date9.);
         %put &date;
      %end;
   %mend date_loop;

%date_loop(01jul2015,01feb2016)
```

Program Description

Begin the macro definition with two parameters.

```
%macro date_loop(start,end);
```

Use the %LET statement to change the value of the macro variable named START. The INPUTN function is used so that the ANYDTDTE informat can be used on the value that is passed to macro variable START. Using the ANYDTDTE informat causes the value of &START to be a SAS formatted date.

```
   %let start=%sysfunc(inputn(&start,anydtdte9.));
```

Use the %LET statement to change the value of the macro variable named END. The INPUTN function is used so that the ANYDTDTE informat can be used on the value that is passed to macro variable END. Using the ANYDTDTE informat causes the value of &END to be a SAS formatted date.

```
%let end=%sysfunc(inputn(&end,anydtdte9.));
```

Use the %LET statement to create a macro variable named DIF. The INTCK function is used to return the number of months between &START and &END. The MONTH function is used as the first argument to retrieve the month interval. When using this function within the macro facility, quotation marks are not used around MONTH, like you would in the DATA step.

```
%let dif=%sysfunc(intck(month,&start,&end));
```

The %DO statement is used to loop through the number of months (&DIF) between &START and &END.

```
  %do i=0 %to &dif;
```

Use the %LET statement to create a macro variable named DATE. The INTNX function is used to increment the &START date by MONTH. The fourth argument, `B`, specifies the alignment. The `B` argument specifies that the returned date or datetime value is aligned to the beginning of the interval. The second argument to %SYSFUNC is the format DATE9, which is applied to the value that is returned from the function INTNX. The %PUT statement is used to write the date value to the log.

```
   %let date=%sysfunc(intnx(month,&start,&i,b),date9.);
   %put &date;
```

End the %DO block.

```
  %end;
```

End the macro.

```
%mend date_loop;
```

Invoke the macro by passing the starting date and the ending date parameters.

```
%date_loop(01jul2015,01feb2016)
```

Log

The following is written to the SAS log:

```
01JUL2015
01AUG2015
01SEP2015
01OCT2015
01NOV2015
01DEC2015
01JAN2016
01FEB2016
```

Example 11: Using Macro Variables within a CARDS or DATALINES Statement

Details

Macro variables are not allowed within the CARDS or DATALINES statements. This example shows a trick that uses the RESOLVE function to avoid this limitation.

Program

```
%let dog=Golden Retriever;

data example;
   input text $40.;
   textresolved=dequote(resolve(quote(text)));
datalines;
John's &dog
My &dog is a female
That's Amy's &dog puppy
;

proc print;
run;
```

Program Description

Create a macro variable named DOG to reference within DATALINES.

```
%let dog=Golden Retriever;
```

Begin the DATA step.

```
data example;
  input text $40.;
```

When using nested functions, start inside and work your way out. The inner most function is QUOTE. It is used to add double quotation marks to the character value contained within the variable TEXT. The RESOLVE function is used to resolve the value of a text expression during the DATA step execution. In this case, the RESOLVE function returns the value of the DATA step variable TEXT. The

DEQUOTE function removes matching quotation marks from a character string that begins with a quotation mark. Then the DEQUOTE function deletes all characters to the right of the closing quotation mark. The final value is assigned to the variable named TEXTRESOLVED.

```
    textresolved=dequote(resolve(quote(text)));
```

Use the DATALINES statement to create the data.

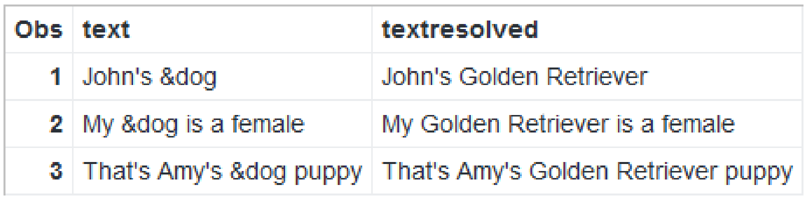
```
  datalines;
  John's &dog
  My &dog is a female
  That's Amy's &dog puppy
  ;
```

Use PROC PRINT to display the results.

```
 proc print;
  run;
```

Output

Output A13.3 *PRINT Procedure Output*



Obs	text	textresolved
1	John's &dog	John's Golden Retriever
2	My &dog is a female	My Golden Retriever is a female
3	That's Amy's &dog puppy	That's Amy's Golden Retriever puppy

Example 12: Print a Note to the SAS Log if a Data Set Is Empty

Details

This macro uses the PRINT procedure to print a data set if it contains observations. If there are no observations, then a note is written to the SAS log.

Program

```
data one;
   x=1;
run;

data two;
   stop;
run;

%macro drive(dsn);
   %let dsid=%sysfunc(open(&dsn));
   %if &dsid ne 0 %then %do;
      %let cnt=%sysfunc(attrn(&dsid,nlobs));
      %let rc=%sysfunc(close(&dsid));
      %if &cnt ne 0 %then %do;
         proc print data=&dsn;
            title "This is data from data set &dsn";
         run;
         title;
      %end;
      %else %do;
         %put NOTE: Data set &dsn is empty.;
      %end;
    %end;
    %else %put &dsn cannot be open.;
%mend drive;

%drive(one)
%drive(two)
```

Program Description

Create a SAS data set containing an observation.

```
data one;
   x=1;
run;
```

Create an empty data set.

```
data two;
   stop;
run;
```

Begin the macro definition with one parameter.

```
%macro drive(dsn);
```

Use the OPEN function to open the data set that is passed to the macro. Use the %IF condition to make sure the data set opened successfully.

```
   %let dsid=%sysfunc(open(&dsn));
   %if &dsid ne 0 %then %do;
```

Use the ATTRN function along with the NLOBS argument to determine the number of observations in the data set that was opened in the OPEN function. Place this value into a macro variable named CNT.

```
      %let cnt=%sysfunc(attrn(&dsid,nlobs));
```

Use the CLOSE function to close the data set.

```
      %let rc=%sysfunc(close(&dsid));
```

Use the %IF statement to check the value in &CNT. If &CNT is not equal to `0`, then the data set contains observations and PROC PRINT is executed.

```
      %if &cnt ne 0 %then %do;
         proc print data=&dsn;
            title "This is data from data set &dsn";
         run;
         title;
      %end;
```

If the %IF statement is false, then the data set is empty and a %PUT statement writes a note to the SAS log stating that the data set is empty.

```
      %else %do;
         %put NOTE: Data set &dsn is empty.;
      %end;
    %end;
```

If data set could not be opened, %ELSE is executed and uses %PUT to write a note to the log.

```
    %else %put &dsn cannot be open.;
```

End the macro.

```
%mend drive;
```

Invoke the macro by passing in the data set name.

```
%drive(one)
```

Here are the results for `%drive(one)`.

Invoke the macro by passing in the data set name.

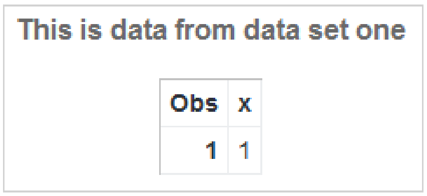
```
%drive(two)
```

Here are the results for `%drive(two)`.

Output

Output: %drive(one)

Output A13.4 *PRINT Statement Output for Data Set One*



This is data from data set one

Obs	x
1	1

Output: %drive(two)

The following note is written to the SAS log:

```
NOTE: Data set two is empty.
```

Example 13: Create a Quoted List Separated by Spaces

Details

This example illustrates how to create a space-separated list that is quoted using the techniques of CALL SYMPUTX and CALL EXECUTE routines.

Program

```
data one;
input empl $;
datalines;
```

```
12345
67890
45678
;

%let list=;
 data _null_;
 set one;
   call symputx('mac',quote(strip(empl)));
   call execute('%let list=&list &mac');
 run;

 %put &=list;
```

Program Description

Create a data set named One.

```
data one;
 input empl $;
 datalines;
 12345
 67890
 45678
 ;
```

Use the %LET statement to create a macro variable named LIST and set it to a null value.

```
 %let list=;
data _null_;
```

Begin a DATA step and set the data set containing the values to be placed in the list. Use CALL SYMPUTX to create a macro variable named MAC. The QUOTE function is used so that the returned value from the data set variable EMPL is quoted. The STRIP function is used to remove any leading or trailing blanks within the quoted value. This value is placed inside the macro variable MAC. CALL EXECUTE is used to build a %LET statement for each observation in the data set. This will append each value of &MAC onto the previous value of &LIST (&LIST is null the first time).

Note: CALL SYMPUTX and CALL EXECUTE are both execution time call routines.

```
set one;
   call symputx('mac',quote(strip(empl)));
   call execute('%let list=&list &mac');
 run;
```

Use %PUT statement to write the contents of macro variable LIST to the SAS log.

```
%put &=list;
```

Log Output

```
LIST="12345" "67890" "45678"
```

Example 14: Delete a File If It Exists

Details

This example uses the FDELETE function to delete a file.

Program

```
%macro check(file);
%if %sysfunc(fileexist(&file)) ge 1 %then %do;
   %let rc=%sysfunc(filename(temp,&file));
   %let rc=%sysfunc(fdelete(&temp));
%end;
%else %put The file &file does not exist;
%mend check;

%check(c:\test.txt)
```

Program Description

Begin the macro definition with one parameter.

```
%macro check(file);
```

Use the FILEEXIST function to determine if the file exists. A nonzero value is returned if the file exists. If the file exists, execute the code within the %DO block. If a zero is returned, meaning that the file does not exist, the %ELSE section is executed.

```
%if %sysfunc(fileexist(&file)) ge 1 %then %do;
```

Use the FILENAME function to associate the file that is passed to the macro (&FILE) with a system-generated fileref that is stored in a macro variable named TEMP.

```
%let rc=%sysfunc(filename(temp,&file));
```

Use the FDELETE function to remove the file that is associated with the fileref stored in macro variable TEMP.

```
%let rc=%sysfunc(fdelete(&temp));
```

End the %IF condition above.

```
%end;
```

If the %IF condition is false, use %PUT to write a note to the log stating that the file does not exist.

```
%else %put The file &file does not exist;
```

End the macro.

```
%mend check;
```

Invoke the macro passing in the full path and file to delete.

```
 %check(c:\test.txt)
```

Example 15: Retrieve the File Size, Create Time, and Last Modified Date of an External File

Details

This example uses the FOPEN and FINFO functions to retrieve certain attributes from an external file.

Program

```
%macro FileAttribs(filename);
  %local rc fid fidc Bytes CreateDT ModifyDT;
  %let rc=%sysfunc(filename(onefile,&filename));
  %let fid=%sysfunc(fopen(&onefile));
     %if &fid ne 0 %then %do;
```

```
      %let Bytes=%sysfunc(finfo(&fid,File Size (bytes)));
      %let CreateDT=%sysfunc(finfo(&fid,Create Time));
      %let ModifyDT=%sysfunc(finfo(&fid,Last Modified));
      %let fidc=%sysfunc(fclose(&fid));
      %let rc=%sysfunc(filename(onefile));
      %put NOTE: File size of &filename is &bytes bytes;
      %put NOTE- Created &createdt;
      %put NOTE- Last modified &modifydt;
         %end;
            %else %put &filename could not be opened.;
    %mend FileAttribs;

    %FileAttribs(c:\aaa.txt)
```

Program Description

Begin the macro definition within one parameter.

```
%macro FileAttribs(filename);
```

Use the %LOCAL statement to make sure all macro variables created are local to the macro FILEATTRIBS.

```
  %local rc fid fidc Bytes CreateDT ModifyDT;
```

Use the FILENAME function to associate the fileref of ONEFILE with the file that is passed to the macro (&FILENAME). The macro variable RC will contain a 0 if the operation was successful; not 0 if unsuccessful.

```
%let rc=%sysfunc(filename(onefile,&filename));
```

Use the FOPEN function to open the file that was set up in the FILENAME function above. The macro variable FID contains a file identifier value that is used to identify the open file to other functions.

```
%let fid=%sysfunc(fopen(&onefile));
```

Use the %IF condition to check that the file opened successfully.

```
      %if &fid ne 0 %then %do;
```

Use the FINFO function to retrieve the size, creation time, and last modified date. The first argument to this function is the macro variable created from the FOPEN function that holds an identifier to the file. Three macro variables are created: BYTES, CREATEDT, and MODIFYDT.

```
  %let Bytes=%sysfunc(finfo(&fid,File Size (bytes)));
  %let CreateDT=%sysfunc(finfo(&fid,Create Time));
  %let ModifyDT=%sysfunc(finfo(&fid,Last Modified));
```

Use the FCLOSE function to close the file.

```
%let fidc=%sysfunc(fclose(&fid));
```

Use the FILENAME function without a second argument to deassign the fileref of ‘onefile’.

```
  %let rc=%sysfunc(filename(onefile));
```

Use the %PUT statement to write the macro variable values to the log.

```
  %put NOTE: File size of &filename is &bytes bytes;
  %put NOTE- Created &createdt;
  %put NOTE- Last modified &modifydt;
```

End the %DO block. If the file could not be opened, the %ELSE is executed and a note is written to the log.

```
   %end;
      %else %put &filename could not be opened.;
```

End the macro definition. Invoke the macro by passing in the full pathname.

```
%mend FileAttribs;

%FileAttribs(c:\aaa.txt)
```

Example 16: Delete All User-defined Macro Variables

Details

This example retrieves all the user-defined global macro variables from the SASHELP.VMACRO view and uses a programming technique with the %SYMDEL function to delete these macro variables.

If running in SAS Enterprise Guide 4.3 or higher, this example also deletes the global macro variable SASWORKLOCATION. A modification can be made to the IF statement to keep this from happening. For more information about the modification, see Example Code A5.1 on page 575.

Program

```
%macro delvars;
  data vars;
    set sashelp.vmacro;
  run;

  data _null_;
    set vars;
    temp=lag(name);
    if scope='GLOBAL' and substr(name,1,3) ne 'SYS' and temp ne name
then
      call execute('%symdel '||trim(left(name))||';');
  run;

%mend delvars;

%delvars
```

Program Description

Begin the macro definition.

```
%macro delvars;
```

Create a data set called Vars that reads the SASHELP.VMACRO view that contains information about currently defined macro variables. A data set is created from SASHELP.VMACRO to avoid a macro symbol table lock.

```
  data vars;
    set sashelp.vmacro;
  run;
```

Within a DATA step, the data set Vars that contains the macro variable names is referenced in the SET statement. The LAG function is used to retrieve the previous value of the name and then assigns it into a data set variable called Temp. Long macro variables will span across multiple observations within the view. This technique is used to assure the unique values.

```
  data _null_;
    set vars;
    temp=lag(name);
```

Use an IF statement to make sure the following occurs:

- the scope of the macro variable is GLOBAL
- the first three characters do not begin with SYS since macro variables beginning with 'SYS' are reserved for use by SAS.
- Temp is not equal to the previous value

If these three items are true, use the CALL EXECUTE routine to build the %SYMDEL statement with the name of the macro variable to be deleted. End the DATA step with a RUN statement. Invoke the macro.

```
    if scope='GLOBAL' and substr(name,1,3) ne 'SYS' and temp ne name then
      call execute('%symdel '||trim(left(name))||';');

run;

%delvars
```

TIP If you are a SAS Enterprise Guide 4.3 user and you want to preserve the SASWORKLOCATION macro variable, replace the previous IF statement in the DATA step with the IF statement in Example Code A5.1 on page 575.

Example Code A13.1 *Modification for SAS Enterprise Guide 4.3 Users*

```
    if scope='GLOBAL' and substr(name,1,3) ne 'SYS' and temp ne name
and
      upcase(name) ne 'SASWORKLOCATION' then
      call execute('%symdel '||trim(left(name))||';');
```