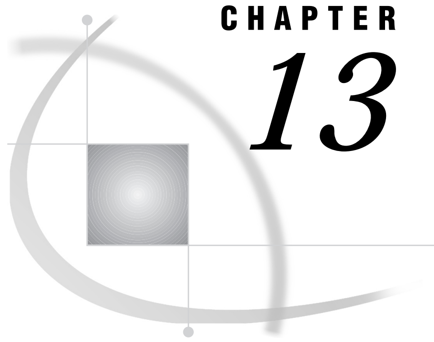


CHAPTER 13

Macro Language Dictionary

%BQUOTE and %NRBQUOTE

Mask special characters and mnemonic operators in a resolved value at macro execution

Type: Macro quoting functions

See also:
"%QUOTE and %NRQUOTE" on page 213
"%SUPERQ" on page 226

Syntax

%BQUOTE (*character string* | *text expression*)

%NRBQUOTE (*character string* | *text expression*)

Details The %BQUOTE and %NRBQUOTE functions mask a character string or resolved value of a text expression during execution of a macro or macro language statement. They mask the following special characters and mnemonic operators:

```
' " ( ) + − * / < > = ¬ ^ ~ ; , blank
AND OR NOT EQ NE LE LT GE GT
```

In addition, %NRBQUOTE masks

```
& %
```

%NRBQUOTE is most useful when the resolved value of an argument may 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.

Tip

You can use %BQUOTE and %NRBQUOTE for all execution-time macro quoting because they mask all characters and mnemonic operators that can be interpreted as elements of macro language. Quotation marks ('") and parentheses (( )) that do not have a match do not have to be marked.

For a description of quoting in SAS macro language, see Chapter 7, "Macro Quoting," in *SAS Macro Language: Reference*.

Comparisons

- %BQUOTE and %NRBQUOTE do not require that you mark quotation marks and parentheses that do not have a match. However, the %QUOTE and %NRQUOTE functions do.
- %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 specfied macro variable. %NRBQUOTE does attempt to resolve such references.

Example

Example 1: 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')
```

Executing this program writes to the log:

```
FILE "myfile"
FILE 'myfile'
```

CMDMAC

Controls command-style macro invocation

Type: System option

Can be specified in:
Configuration file
OPTIONS window
OPTIONS statement
SAS invocation

Default: NOCMDMAC

Syntax

CMDMAC | NOCMDMAC

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 it finds with the same name as the first word in the windowing environment command that was issued. This can produce unexpected results. △

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 parmameter-value-1 parmameter-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 those defined as command-style macros.

Tip

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.

%CMPRES and %QCMPRES

Compress multiple blanks and remove leading and trailing blanks

Type: Autocall macros
Requires: MAUTOSOURCE system option

Syntax

%CMPRES (*text* | *text expression*)

%QCMPRES (*text* | *text expression*)

Note: Autocall macros are included in a library supplied by SAS Institute. This library may not be installed at your site or may 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 SAS Software Consultant. For more information, see Chapter 9, "Storing and Reusing Macros," in *SAS Macro Language: Reference*. △

Details 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
```

Examples

Example 1: Removing Unnecessary Blanks with %CMPRES

```
%macro createft;
   %let footnote="The result of &x &op &y is %eval(&x &op &y).";
   footnote1 &footnote;
   footnote2 %cmpres(&footnote);
%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 generates two footnote statements.

```
FOOTNOTE1 "The result of 5 + __________10 is __________15.";
FOOTNOTE2 "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 statement writes the line

```
QCMPRES: %eval(&x + &y)
CMPRES: 15
```

%* *comment*

Designates comment text

Type: Macro statement

Restriction: Allowed in macro definitions or open code

Syntax

%*_comment_;

comment
is a descriptive message of any length.

Details The macro comment statement is useful for describing macro code. Text from a macro comment statement is not constant text and is not stored in a compiled macro. Because a semicolon ends the comment statement, the comment cannot contain internal semicolons unless the internal semicolons are enclosed in quotation marks or a macro quoting function. Macro comments are not recognized when they are enclosed in quotation marks.

Quotation marks within a macro comment must match.

Comparisons

Macro comments and SAS comments in the form `*comment;` are complete statements. Consequently, they are processed by the tokenizer and cannot contain semicolons or unmatched quotation marks. SAS comments in the form `*comment;` are stored as constant text in a compiled macro.

SAS comments in the form `/*comment*/` are not tokenized, but are processed as a string of individual characters. These comments can appear anywhere a single blank can appear and can contain semicolons or unmatched quotation marks. SAS comments in the form `/*comment*/` are not stored in a compiled macro.

Example

Example 1: Contrasting Comment Types This code defines and invokes the macro VERDATA, which checks for data errors. It contains a macro comment and comments in the form `/*comment*/` and `*comment;`.

```
%macro verdata(in);
   %if %length(&in) > 0 %then %do;
```

```
          %* infile given;
      data check;
            /* Jim's data */
         infile &in;
         input x y z;
            * check data;
         if x<0 or y<0 or z<0 then list;
      run;
   %end;
   %else %put Error: No infile specified;
%mend verdata;

%verdata(ina)
```

When you execute VERDATA, the macro processor generates the following:

```
DATA CHECK;
   INFILE INA;
   INPUT X Y Z;
      * CHECK DATA;
   IF X<0 OR Y<0 OR Z<0 THEN LIST;
RUN;
```

%COMPSTOR

Compiles macros and stores them in a catalog in a permanent SAS library

Type: Autocall macro

Requires: MAUTOSOURCE system option

Syntax

%COMPSTOR (PATHNAME=*SAS-data-library*)

SAS-data-library
is the physical name of a SAS data 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.

Note: Autocall macros are included in a library supplied by SAS Institute. This library may not be installed at your site or may 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 SAS Software Consultant. For more information, see Chapter 9 in *SAS Macro Language: Reference*. △

Details The COMPSTOR macro compiles the following autocall macros in a SAS catalog named SASMACR in a permanent SAS data library. This saves the overhead of compiling these macros when they are called for the first time in a SAS session. You

can use the COMPSTOR macro as an example of how to create compiled stored macros. For more information on the SAS supplied autocall macros or about using stored compiled macros, see Chapter 9 in *SAS Macro Language: Reference*.

%CMPRES

%DATATYP

%LEFT

%QCMPRES

%QLEFT

%QTRIM

%TRIM

%VERIFY

%DATATYP

Returns the data type of a value

Type: Autocall macro

Requires: MAUTOSOURCE system option

Syntax

%DATATYP (*text* | *text expression*)

Note: Autocall macros are included in a library supplied by SAS Institute. This library may not be installed at your site or may 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 SAS Software Consultant. For more information, see Chapter 9 in *SAS Macro Language: Reference*. △

Details The DATATYP macro returns a value of **NUMERIC** when an argument consists of digits and, optionally, 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

Example 1: 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 as:

```
%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 as:

```
%add(0c1x, 12)
```

The macro then writes this message to the SAS log:

```
Error:  Addition requires numbers.
```

%DISPLAY

Displays a macro window

Type: Macro statement

Restriction: Allowed in macro definitions or open code

See also: "%WINDOW" on page 279

Syntax

%DISPLAY *window*<*.group*><NOINPUT><BLANK>
<BELL><DELETE>;

***window* <*.group*>**
names the window and group of fields to be displayed. If the window has more than one group of fields, give the complete *window.group* specification; if a window contains a single unnamed group, specify only *window*.

NOINPUT
specifies that you cannot input values into fields displayed in the window. If you omit the NOINPUT option, you can input values into unprotected fields displayed in the window. Use the NOINPUT option when the %DISPLAY statement is inside a macro definition and you want to merge more than one group of fields into a single display. Using NOINPUT in a particular %DISPLAY statement causes the group displayed to remain visible when later groups are displayed.

BLANK
clears the display. Use the BLANK option to prevent fields from a previous display from appearing in the current display. This option is useful only when the %DISPLAY statement is inside a macro definition and when it is part of a *window.group* specification. When the %DISPLAY statement is outside a macro definition, the display is cleared automatically after the execution of each %DISPLAY statement.

BELL
rings the terminal's bell, if available, when the window is displayed.

DELETE
deletes the display of the window after processing passes from the %DISPLAY statement on which the option appears. DELETE is useful only when the %DISPLAY statement is inside a macro definition.

Details You can display only one group of fields in each execution of a %DISPLAY statement. If you display a window containing any unprotected fields, enter values into any required fields and press ENTER to remove the display.

If a window contains only protected fields, pressing ENTER removes the display. While a window is displayed, you can use commands and function keys to view other windows, change the size of the current window, and so on.

%DO

Begins a %DO group

Type: Macro statement

Restriction: Allowed in macro definitions only

See also: "%END" on page 170

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

Example 1: 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, Iterative

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 also: “%END” on page 170

Syntax

%DO *macro-variable=start* %TO *stop* <%BY *increment*>;
text and macro language statements

%END;

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
stop
specify integers or macro expressions that generate integers to control the number of times the portion of the macro between the iterative %DO and %END statements is processed.

The first time the %DO group iterates, *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 included by *start* and *stop*.

increment
specifies an integer (other than 0) or a macro expression that generates an 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

Example 1: 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

Executes a section of a macro repetitively until a condition is true

Type: Macro statement

Restriction: Allowed in macro definitions only

See also: "%END" on page 170

Syntax

%DO %UNTIL (*expression*);
text and macro language statements

%END;

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

Example 1: 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=BLOCK 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 chart;type sex / group=dept;
      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;
```

When 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

Executes a section of a macro repetitively while a condition is true

Type: Macro statement

Restriction: Allowed in macro definitions only

See also: "%END" on page 170

Syntax

%DO %WHILE (*expression*);
text and macro program statements

%END;

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

Example 1: 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

Ends a %DO group

Type: Macro statement

Restriction: Allowed in macro definitions only

Syntax

%END;

Example

Example 1: 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
```

%EVAL

Evaluates arithmetic and logical expressions using integer arithmetic

Type: Macro evaluation function

See also: “%SYSEVALF” on page 246

Syntax

%EVAL (*arithmetic or logical expression*)

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. Refer to “The SORT PROCEDURE” in the *SAS Procedures Guide* for more information on operating environment sort sequences.

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,” in *SAS Macro Language: Reference*.

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;
```

Submitting these statements prints 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 implicitly 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)
```

Executing this program 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
```

Example 3: Evaluating Logical Expressions Macro COMPARE comapres two numbers.

```
%macro compare(first,second);
   %if &first>&second %then %put &first > &second;
   %else %if &first=&second %then %put &first = &second;
   %else %put &first<&second;
%mend compare;

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

Executing this program writes these lines to the SAS log:

```
1 < 2
-1 < 0
```

EXECUTE

Resolves its argument and executes the resolved value at the next step boundary

Type: DATA step routine

Syntax

CALL EXECUTE (*argument*);

argument

can be

- 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

```
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

```
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

```
call execute('%sales('||month||')');
```

Details If an EXECUTE routine argument is a macro invocation or resolves to one, the macro executes immediately. However, any SAS statements produced by the EXECUTE routine do not execute until *after* the step boundary has been passed.

Note: Because macro references execute immediately and SAS statements do not execute until after a step boundary, you cannot use CALL EXECUTE to invoke a macro that contains references for macro variables that are created by CALL SYMPUT in that macro. See Chapter 8, "Interfaces with the Macro Facility," in *SAS Macro Language: Reference*, for an example. △

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 The following DATA step uses CALL EXECUTE to execute a macro only if the DATA step writes at least one observation to the temporary data set.

```
%macro overdue;
   proc print data=late;
      title "Overdue Accounts As of &sysdate";
   run;
%mend overdue;

data late;
   set sasuser.billed end=final;
   if datedue<=today()-30 then
      do;
         n+1;
         output;
      end;
   if final and n then call execute('%overdue');
run;
```

Example 2: Passing DATA Step Values Into a Parameter List CALL EXECUTE passes the value of the DATE variable in the DATES data set to macro REPT for its DAT parameter, the value of the VAR1 variable in the REPTDATA data set for its A parameter, and REPTDATA as the value of its DSN parameter. After the DATA _NULL_ step finishes, three PROC GCHART statements are submitted, one for each of the three dates in the DATES data set.

```
data dates;
   input date $;
cards;
10nov97
11nov97
12nov97
;

data reptdata;
   input date $ var1 var2;
cards;
10nov97 25 10
10nov97 50 11
11nov97 23 10
11nov97 30 29
12nov97 33 44
12nov97 75 86
;

%macro rept(dat,a,dsn);
   proc chart data=&dsn;
        title "Chart for &dat";
        where(date="&dat");
        vbar &a;
   run;
%mend rept;

data _null_;
   set dates;
   call execute('%rept('||date||','||'var1,reptdata)');
run;
```

%GLOBAL

Creates macro variables that are available during the execution of an entire SAS session

Type: Macro statement

Restriction: Allowed in macro definitions or open code

See also: "%LOCAL" on page 191

Syntax

%GLOBAL *macro-variable(s)*;

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.

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 SAS 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.

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

Example 1: 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;
```

The SAS System sees the following:

```
PROC PRINT;
   VAR TEST1-TEST50;
   TITLE "Analysis of Tests 1-50";
RUN;
```

%GOTO

Branches macro processing to the specified label

Type: Macro statement

Alias: %GO TO

Restriction: Allowed in macro definitions only

See also: “%*label*” on page 187

Syntax

%GOTO *label*;

label

is either the name of the label in the macro 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*.*

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

* A computed %GOTO contains `%` or `&` and resolves to a label.

statement cannot cause execution to branch to a point inside an iterative %DO, %DO %UNTIL, or %DO %WHILE loop that is not currently executing.

Example

Example 1: 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 sucessfully.;
%exit: %mend check;
```

%IF-%THEN/%ELSE

Conditionally process a portion of a macro

Type: Macro statement

Restriction: Allowed in macro definitions only

Syntax

%IF *expression* **%THEN** *action*;

<**%ELSE***action*;>

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, see Chapter 6 in *SAS Macro Language: Reference*.

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;
  ```

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 uses the sort sequence of the host operating system for the comparison. Refer to "The SORT PROCEDURE" in the *SAS Procedures Guide* for more information on host sort sequences.

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 Chapter 2, “SAS Programs and Macro Processing,” and Chapter 6 in *SAS Macro Language: Reference*.

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;
```

When 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;
```

When 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;
```

IMPLMAC

Controls statement-style macro invocation

Type: System option

Can be specified in:
Configuration file
OPTIONS window
OPTIONS statement
SAS invocation

Default: NOIMPLMAC

Syntax

IMPLMAC | NOIMPLMAC

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 it finds with the same name as the first word in the SAS statement that was submitted. This can produce unexpected results. △

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 those defined as 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 and unexpected results can occur. △

Tip

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.

%INDEX

Returns the position of the first character of a string

Type: Macro function

Syntax

%INDEX (*source,string*)

source

is a character string or text expression.

string
is a character string or text expression.

Details The %INDEX function searches *source* for the first occurrence of *string* and returns the position of its first character. If *string* is not found, the function returns 0.

Example

Example 1: 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..;
```

Executing these statements writes to the SAS log:

```
V appears at position 3.
```

%INPUT

Supplies values to macro variables during macro execution

Type: Macro statement

Restriction: Allowed in macro definitions or open code

See also:
“%PUT” on page 208
“%WINDOW” on page 279
“SYSBUFFR” on page 238

Syntax

%INPUT< *macro-variable(s)*>;

no argument
specifies that all text entered is assigned to the automatic macro variable SYSBUFFR.

macro-variable(s)
is the name of a macro variable or a macro text expression that produces a macro variable name. The %INPUT statement can contain any number of variable names separated by blanks.

Details The macro processor interprets the line submitted immediately after a %INPUT statement as the response to the %INPUT statement. That line can be part of an interactive line mode session, or it can be submitted from within the PROGRAM EDITOR window during a windowing environment session.

When a %INPUT statement executes as part of an interactive line mode session, the macro processor waits for you to enter a line containing values. In a windowing environment session, the macro processor does *not* wait for you to input values. Instead, it simply reads the next line that is processed in the program and attempts to assign variable values. Likewise, if you invoke a macro containing a %INPUT statement in open code as part of a longer program in a windowing environment, the macro processor reads the next line in the program that follows the macro invocation. Therefore, when you submit a %INPUT statement in open code from a windowing environment, make sure that the line that follows a %INPUT statement or a macro invocation that includes a %INPUT statement contains the values you want to assign.

When you name variables in the %INPUT statement, the macro processor matches the variables with the values in your response based on their positions; that is, the first value you enter is assigned to the first variable named in the %INPUT statement, the second value is assigned to the second variable, and so on.

Each value to be assigned to a particular variable must be a single word or a string enclosed in quotation marks. To separate values, use blanks. After all values have been matched with macro variable names, excess text becomes the value of the automatic macro variable SYSBUFFR.

Example

Example 1: Assigning a Response to a Macro Variable In an interactive line mode session, the following statements display a prompt and assign the response to the macro variable FIRST:

```
%put Enter your first name:;
%input first;
```

INTO

Assigns values produced by PROC SQL to macro variables

Type: SELECT statement, PROC SQL

Syntax

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

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*

specifies one or more macro variables. Leading and trailing blanks are not trimmed from values before they are stored in macro variables. For example,

```
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. For example,

```
select style, sqfeet
   into :type1 - :type4 notrim, :size1 - :size3
   from sasuser.houses;
```

:*macro-variable* SEPARATED BY '*character(s)*' <NOTRIM>

specifies one macro variable to contain all the values of a column. Values in the list are separated by *character(s)*. 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 on the SELECT statement to store only the unique column (variable) values. For example,

```
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 Chapter 5, "Scope of Macro Variables," in *SAS Macro Language: Reference*.

Values assigned by the INTO clause use the BEST12. format.

Comparisons

In the SQL procedure, the INTO clause performs a role similar to the SYMPUT routine.

Examples

Example 1: Storing Column Values in Explicitly-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;
```

```
%let type=&type;
%let size=&size;

%put The first row contains a &type with &size square feet.;
```

Executing this program writes 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;

%macro putit;
   %do i=1 %to 4;
      %put Row&i: Type=**&&type&i**   Size=**&&size&i**;
   %end;
%mend putit;

%putit
```

Executing this program writes these lines 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;

%put Types of houses=&types.;
```

Executing this program writes this line to the SAS log:

```
Types of houses=CONDO, RANCH, SPLIT, TWOSTORY
```

%KEYDEF

Assigns a definition to or identifies the definition of a function key

Type: Macro statement

Restriction: Allowed in macro definitions or open code

Syntax

%KEYDEF *key-name*< *definition*>;

key-name

is the name of any function key on your terminal, such as F1. The maximum length for *key-name* is the same as for the KEYDEF statement in the base SAS language. If you use only the *key-name* argument, SAS issues a message identifying the definition of that function key. If the name of a function key contains more than one word or if it contains special characters, then enclose the name in quotation marks.

The following examples illustrate using the %KEYDEF statement:

- ```
  %keydef f1;
  ```
- ```
  %keydef pf12;
  ```
- ```
  %keydef 'SHF F6';
  ```
- ```
  %keydef '^a';
  ```

Operating Environment Information: The names of function keys vary by operating environment and hardware. For details on the KEYS window, see the SAS documentation for your operating environment. △

definition

is the text (up to 80 characters long) that you want to assign to the function key. The definition can also be within single or double quotation marks.

If the definition is not in quotation marks, any macro invocations or macro variables are resolved, and the definition is converted to uppercase. However, a definition that is not in quotation marks cannot contain semicolons unless they are enclosed in a macro quoting function. If the definition is more than 80 characters long, it is truncated to 80 characters without a warning or error message.

If the definition is enclosed in double quotation marks, macro references or macro variable references are resolved. If the resolved definition is more than 80 characters long, it is truncated to 80 characters without a warning or error message.

If the definition is enclosed in single quotation marks, macro references or macro variables are not resolved until the function key is pressed. To circumvent the 80 character limit on *definition*, you can use a macro invocation or macro variable reference.

The value of *definition* is

command | ~*text*

command

is one or more windowing environment commands. When you press the function key, the assigned command or commands are submitted within the current window.

~text
is any text that you want to insert after the current cursor position. When you press the function key, the assigned text is inserted after the cursor position in any field of any window that accepts input. The tilde symbol does not become part of the inserted text.

Details Use the %KEYDEF statement only in a SAS windowing environment session. Function key definitions you assign with %KEYDEF remain in effect for the duration of your current SAS session or until you change them again during the session. To save function key settings from one session to the next, use the KEYS window or place the %KEYDEF statement in an autoexec file.

Example

Example 1: Sample %KEYDEF Statements In these examples the values f1 through f6 represent the names of function keys.

- ```
  %keydef f1 %ckclear;   /* assigns text generated by a macro to */
                         /* a function key (note no quotes)    */
  ```
- ```
  %keydef f1 "%ckclear"; /* assigns text generated by a macro to a */
                         /* function key (note the double quotes) */
  ```
- ```
  %keydef f2 '%ckclear'; /* assigns a macro call to a function key  */
                         /* (note the single quotes) */
  ```
- ```
  %keydef f3 'pgm; submit "proc print data=hotdog; run;"';
  ```
- ```
  %keydef f4 "%app";     /* Appends transaction file to master */
  ```
- ```
  %keydef f5 '%printit'; /* Prints selected variables */'
  ```
- ```
  %keydef f6 ~(obs=10);  /* Add OBS=10 DATA set option */
  ```
- ```
  %macro mykeys;
               %keydef f2 "clear log;clear output";
               %keydef f4 rfind;
          %mend mykeys;
  ```

%label

Identifies the destination of a %GOTO statement

Type: Macro statement

Restriction: Allowed in macro definitions only

See also: "%GOTO" on page 176

Syntax

%label: *macro-text*

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

Example 1: 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.

```
%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=10);     /* Use % here */
      run;
%mend info;

%info(short)
```

Invoking the macro INFO with TYPE equal to **short** generates these statements:
```

```
PROC PRINT DATA=_LAST_(OBS=10);
   RUN;
```

%LEFT and %QLEFT

Left-align an argument by removing leading blanks

Type: Autocall macro
Requires: MAUTOSOURCE system option

Syntax

%LEFT(*text* | *text expression*)

%QLEFT(*text* | *text expression*)

Note: Autocall macros are included in a library supplied by SAS Institute. This library may not be installed at your site or may 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 SAS Software Consultant. For more information, see Chapter 9, "Storing and Reusing Macros" in *SAS Macro Language: Reference*. △

Details 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 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
```

Example

Example 1: 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 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   *
```

%LENGTH

Returns the length of a string

Type: Macro function

Syntax

%LENGTH (*character string* | *text expression*)

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.

Example

Example 1: 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).;
```

Executing these statements writes 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.
```

%LET

Creates a macro variable and assigns it a value

Type: Macro statement

Restriction: Allowed in macro definitions or open code

See also: “%STR and %NRSTR” on page 221

Syntax

%LET *macro-variable* =< *value*>;

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 named in the %LET statement already exists, the %LET statement changes the value. A %LET statement can define only one macro variable at a time.

Example

Example 1: Sample %LET Statements These examples illustrate several %LET statement:

```
%macro title(text,number);
    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 macro generates the following statements:

```
TITLE1 "The History of Genetics";
TITLE2 "Genetics Today";
TITLE3 "The Future of Genetics";
```

%LOCAL

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 also: “%GLOBAL” on page 175

Syntax

%LOCAL_macro-variable(s)_;

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.

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.

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

Example 1: 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
```

%LOWCASE and %QLOWCASE

Change uppercase characters to lowercase

Type: Autocall macros

Requires: MAUTOSOURCE system option

Syntax

%LOWCASE (*text* | *text expression*)

%QLOWCASE (*text* | *text expression*)

Note: Autocall macros are included in a library supplied by SAS Institute. This library may not be installed at your site or may 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 SAS Software Consultant. For more information, see Chapter 9 in *SAS Macro Language: Reference*. △

Details 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 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
```

Example

Example 1: Creating a Title with Initial Letters Capitalized

```
%macro initcaps(title);
   %global newtitle;
   %let newtitle=;
   %let lastchar=;
   %do i=1 %to %length(&title);
      %let char=%qsubstr(&title,&i,1);
      %if (&lastchar=%str( ) or &i=1) %then %let char=%qupcase(&char);
      %else %let char=%qlowcase(&char);
      %let newtitle=&newtitle&char;
      %let lastchar=&char;
   %end;
   TITLE "&newtitle";
%mend;

%initcaps(%str(sales: COMMAND REFERENCE, VERSION 2, SECOND EDITION))
```

Submitting this example generates the following statement:

```
TITLE "Sales: Command Reference, Version 2, Second Edition";
```

%MACRO

Begins a macro definition

Type: Macro statement

Restriction: Allowed in macro definitions or open code

See also:

“%MEND” on page 199
“SYSPBUFF” on page 259

Syntax

%MACRO *macro-name* <(*parameter-list*)></ *option(s)*>;

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 Used in the Macro Facility,” in *SAS Macro Language: Reference*.)

parameter-list

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.

parameter-list can be

<*positional parameter-1*><. . . ,*positional paramter-n*>

<*keyword-parameter*=<*value*> <. . . ,*keyword-parameter-n*=<*value*>>>

positional-parameter-1 <. . . ,*positional-parameter-n*>

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 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 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-n*=<*value*>>

names one or more macro parameters followed by equal signs. Optionally, 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 allows 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. △

option(s)

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 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. 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 useful only when you use the stored compiled macro facility.

PARMBUFF
PBUFF
: 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 and and 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 to indicate the absence of a value list, even if the macro definition contains no parameters.

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.

STORE
: stores the compiled macro as an entry in a SAS catalog in a permanent SAS data library. Use the SAS system option SASMSTORE= to identify a permanent SAS data 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 Chapter 9 in *SAS Macro Language: Reference*.)

Details The %MACRO statement begins the definition of a macro, assigns the macro a name, and optionally 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 also can store a macro in a permanent SAS catalog for future use. However, in Version 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 these 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
  ```

  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 equals 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 allow you to input a different number of arguments each time you invoke it:

```
%macro printz/parmbuff;
   %let num=1;
   %let dsname=%scan(&syspbuff,&num);
   %do %while(&dsname ne);
      proc print data=&dsname;
      run;
      %let num=%eval(&num+1);
      %let dsname=%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:

```
PROC PRINT DATA=PURPLE;
RUN;
PROC PRINT DATA=RED;
RUN;
PROC PRINT DATA=BLUE;
RUN;
PROC PRINT DATA=TEAL;
RUN;
```

MACRO

Controls whether the SAS macro language is available

Type: System option

Can be specified in:
Configuration file
SAS invocation

Default: MACRO

Syntax

MACRO | NOMACRO

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.

MAUTOSOURCE

Controls whether the autocall feature is available

Type: System option

Can be specified in:
Configuration file
OPTIONS window
OPTIONS statement
SAS invocation

Default: MAUTOSOURCE

Syntax

MAUTOSOURCE | NOMAUTOSOURCE

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 SAS 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.

%MEND

Ends a macro definition

Type: Macro statement

Restriction: Allowed in macro definitions only

Syntax

%MEND <*macro-name*>;

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

Example 1: Ending a Macro Definition

```
%macro disc(dsn);
   data &dsn;
      set perm.dataset;
      where month="&dsn";
   run;
%mend disc;
```

MERROR

Controls whether the macro processor issues a warning message when a macro reference cannot be resolved

Type: System option

Can be specified in:

Configuration file

OPTIONS window

OPTIONS statement

SAS invocation

Default: MERROR

Syntax

MERROR | NOMERROR

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:

```
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.

MFILE

Determines whether MPRINT output is routed to an external file

Type: System option

Can be specified in:
Configuration file
OPTIONS window
OPTIONS statement
SAS invocation

Requires: MPRINT option

Default: NOMFILE

See also: "MPRINT" on page 202

Syntax

MFILE | NOMFILE

MFILE
routes output produced by the MPRINT option to an external file. This is useful for debugging.

NOMFILE
does not route MPRINT output to an external file.

Details The MPRINT option must also be in effect to use MFILE, and an external file must be assigned the fileref MPRINT. 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. To use the feature again, you must specify MFILE again and assign the fileref MPRINT to a file that can be accessed.

MLOGIC

Controls whether macro execution is traced for debugging

Type: System option

Can be specified in:

Configuration file

OPTIONS window

OPTIONS statement

SAS invocation

Default: NOMLOGIC

Syntax

MLOGIC | NOMLOGIC

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.

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 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 on macro debugging, see Chapter 10, "Macro Facility Error Messages and Debugging," in *SAS Macro Language: Reference*.

Example

Example 1: Tracing Macro Execution In this example, MLOGIC traces the execution of the macros MKTITLE and RUNPLOT:

```
%macro mktitle(proc,data);
    title "%upcase(&proc) of %upcase(&data)";
%mend mktitle;

%macro runplot(ds);
   %if %sysprod(graph)=1 %then
      %do;
         %mktitle (gplot,&ds)
         proc gplot data=&ds;
            plot style*price
                / haxis=0 to 150000 by 50000;
         run;
         quit;
      %end;
   %else
      %do;
         %mktitle (plot,&ds)
         proc plot data=&ds;
            plot style*price;
         run;
         quit;
      %end;
%mend runplot;

options mlogic;

%runplot(sasuser.houses)
```

Executing this program writes this MLOGIC output to the SAS log:

```
MLOGIC(RUNPLOT):  Beginning execution.
MLOGIC(RUNPLOT):  Parameter DS has value sasuser.houses
MLOGIC(RUNPLOT):  %IF condition %sysprod(graph)=1 is TRUE
MLOGIC(MKTITLE):  Beginning execution.
MLOGIC(MKTITLE):  Parameter PROC has value gplot
MLOGIC(MKTITLE):  Parameter DATA has value sasuser.houses
MLOGIC(MKTITLE):  Ending execution.
MLOGIC(RUNPLOT):  Ending execution.
```

MPRINT

Controls whether SAS statements generated by macro execution are traced for debugging

Type: System option

Can be specified in:

Configuration file

OPTIONS window

OPTIONS statement

SAS invocation

Default: NOMPRINT

See also: "MFILE" on page 200

Syntax

MPRINT | NOMPRINT

MPRINT
Displays SAS statements generated by macro execution. This is useful for debugging macros.

NOMPRINT
Does not display SAS statements generated by macro execution.

Details The MPRINT option displays the text generated by macro execution. Each SAS statement 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. Each statement ends with a semicolon.

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.

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,data);
    title "%upcase(&proc) of %upcase(&data)";
%mend mktitle;

%macro runplot(ds);
   %if %sysprod(graph)=1 %then
      %do;
         %mktitle (gplot,&ds)
         proc gplot data=&ds;
            plot style*price
                 / haxis=0 to 150000 by 50000;
         run;
         quit;
      %end;
   %else
      %do;
         %mktitle (plot,&ds)
         proc plot data=&ds;
            plot style*price;
         run;
         quit;
      %end;
%mend runplot;

options mprint;
```

```
%runplot(sasuser.houses)
```

Executing this program writes this MPRINT output to the SAS log:

```
MPRINT(MKTITLE):   TITLE "GPLOT of SASUSER.HOUSES";
MPRINT(RUNPLOT):   PROC GPLOT DATA=SASUSER.HOUSES;
MPRINT(RUNPLOT):   PLOT STYLE*PRICE / HAXIS=0 TO 150000 BY 50000;
MPRINT(RUNPLOT):   RUN;

MPRINT(RUNPLOT):   QUIT;
```

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 SAS session ends.

```
options mfile mprint;
filename mprint 'debugmac';
```

MRECALL

Controls whether autocall libraries are searched for a member not found during an earlier search

Type: System option

Can be specified in:
Configuration file
OPTIONS window
OPTIONS statement
SAS invocation

Default: NOMRECALL

Syntax

MRECALL | NOMRECALL

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.

MSTORED

Controls whether stored compiled macros are available

Type: System option

Can be specified in:
Configuration file
OPTIONS window
OPTIONS statement
SAS invocation

Default: NOMSTORED

Syntax

MSTORED | NOMSTORED

MSTORED
searches for stored compiled macros in a catalog in the SAS data 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 SAS 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.

MSYMTABMAX=

Specifies the maximum amount of memory available to the macro variable symbol table(s)

Type: System option

Can be specified in:
Configuration file
OPTIONS window
OPTIONS statement
SAS invocation

Syntax

MSYMTABMAX= *n* | *n*K | *n*M | *n*G | MAX

n	specifies the maximum memory available in bytes.
*n*K	specifies the maximum memory available in kilobytes.
*n*M	specifies the maximum memory available in megabytes.
*n*G	specifies the maximum memory available in gigabytes.
MAX	specifies the maximum memory available as the largest integer your operating environment can represent.

Details Once the maximum value is reached, additional macro variables are written out to disk.

The value you specify with the MSYMTABMAX= system option can range from 0 to the largest non-negative 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=

Specifies the maximum size for in-memory macro variable values

Type: System option

Can be specified in:

Configuration file
OPTIONS window
OPTIONS statement
SAS invocation

Syntax

MVARSIZE= *n* | *n*K | *n*M | *n*G MAX

n	specifies the maximum memory available in bytes.
*n*K	specifies the maximum memory available in kilobytes.
*n*M	specifies the maximum memory available in megabytes.
*n*G	specifies the maximum memory available in gigabytes.
MAX	specifies the maximum memory available as the largest integer your operating system can represent.

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 you specify with the MVARSIZE= system option can range from 0 to the largest non-negative integer representable on your host. The default values are host dependent. 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.

%NRBQUOTE

Masks special characters, including & and %, and mnemonic operators in a resolved value at macro execution

Type: Macro quoting function

Syntax

%NRBQUOTE (*character string* | *text expression*)

See “%BQUOTE and %NRBQUOTE” on page 157

%NRQUOTE

Masks special characters, including & and %, and mnemonic operators in a resolved value at macro execution

Type: Macro quoting function

Syntax

%NRQUOTE (*character string* | *text expression*)

See “%QUOTE and %NRQUOTE” on page 213

%NRSTR

Masks special characters, including & and %, and mnemonic operators in constant text during macro compilation

Type: Macro quoting function

Syntax

%NRSTR (*character-string*)

See “%STR and %NRSTR” on page 221

%PUT

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_ | _USER_>;

no argument
places a blank line in the SAS log.

text
is text or a text expression that is written to the SAS log. 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.

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.

USER
describes user-generated global and local macro variables. The scope is identified either as GLOBAL, for global macro variables, or as the name of the macro in which the macro variable is defined.

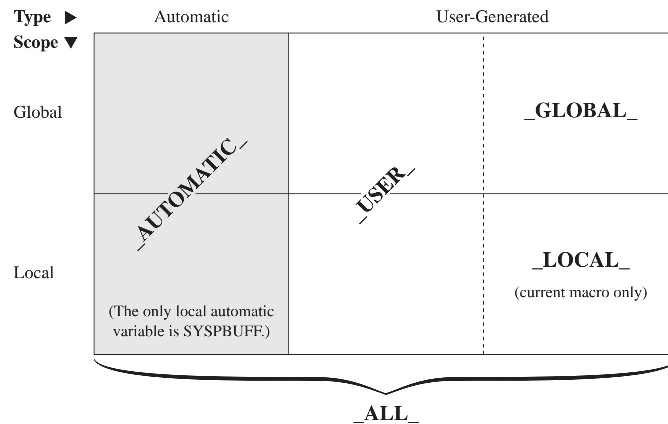
Details When you use the %PUT statement to list macro variable descriptions, the %PUT statement includes only the macro variables that exist at the time the statement executes. The description contains the macro variable's scope, name, and value. Macro variables with null values show only the scope and 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: Within a particular scope, macro variables may appear in any order, and the order may change in different executions of the %PUT statement or different SAS

sessions. Do not write code that depends on locating a variable in a particular position in the list. △

Figure 13.1 on page 209 shows the relationship of these terms.

Figure 13.1 %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, followed immediately by a colon or a hyphen. You may also use the national-language equivalents of these words. When you use a hyphen, the ERROR, NOTE, or WARNING word is blanked out.

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:

```
AUTOMATIC SYSBUFFR
AUTOMATIC SYSCMD
AUTOMATIC SYSDATE 21JUN97
AUTOMATIC SYSDAY Wednesday
AUTOMATIC SYSDEVIC
AUTOMATIC SYSDSN         _NULL_
```

```
AUTOMATIC SYSENV FORE
AUTOMATIC SYSERR 0
AUTOMATIC SYSFILRC 0
AUTOMATIC SYSINDEX 0
AUTOMATIC SYSINFO 0
```

Example 3: Displaying User-Generated Variables This example lists the user-generated macro variables in all referencing environments.

```
%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);

   proc freq data=&name;
      tables &vars;
   run;

   %put FIRST LIST:;
   %put _local_;

   %let firstvar=%scan(&vars,1);

   proc print data=&name;
      where &firstvar ne .;
   run;

   %put SECOND LIST:;
   %put _local_;

%mend analyze;
```

```
%analyze(consumer,car house stereo)
```

In the result, 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 NAME consumer
ANALYZE VARS car house stereo

SECOND LIST:
ANALYZE NAME consumer
ANALYZE VARS car house stereo
ANALYZE FIRSTVAR car
```

%QCMPRES

Compresses multiple blanks, removes leading and trailing blanks, and returns a result that masks special characters and mnemonic operators

Type: Autocall macro

Requires: MAUTOSOURCE system option

Syntax

%QCMPRES (*text* | *text expression*)

See "%CMPRES and %QCMPRES" on page 159

Note: Autocall macros are included in a library supplied by SAS Institute. This library may not be installed at your site or may 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 SAS Software Consultant. For more information, see Chapter 9 in *SAS Macro Language: Reference*. △

%QLEFT

Left-aligns an argument by removing leading blanks and returns a result that masks special characters and mnemonic operators

Type: Autocall macro

Requires: MAUTOSOURCE system option

Syntax

%QLEFT (*text* | *text expression*)

See "%LEFT and %QLEFT" on page 189

Note: Autocall macros are included in a library supplied by SAS Institute. This library may not be installed at your site or may 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 SAS Software Consultant. For more information, see Chapter 9 in *SAS Macro Language: Reference*. △

%QLOWCASE

Changes uppercase characters to lowercase and returns a result that masks special characters and mnemonic operators

Type: Autocall macro

Requires: MAUTOSOURCE system option

Syntax

%QLOWCASE(*text* | *text expression*)

See “%LOWCASE and %QLOWCASE” on page 192

Note: Autocall macros are included in a library supplied by SAS Institute. This library may not be installed at your site or may 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 SAS Software Consultant. For more information, see Chapter 9 in *SAS Macro Language: Reference*. △

%QSCAN

Searches for a word and masks special characters and mnemonic operators

Type: Macro function

Syntax

%QSCAN (*argument*,*n*<,*delimiters*>)

See “%SCAN and %QSCAN” on page 218.

%QSUBSTR

Produces a substring and masks special characters and mnemonic operators

Type: Macro function

Syntax

%QSUBSTR (*argument,position*<,*length*>)

See “%SUBSTR and %QSUBSTR” on page 223

%QSYSFUNC

Executes functions and masks special characters and mnemonic operators

Type: Macro function

Syntax

%QSYSFUNC(*function*(*function-arg-list*)<,*format*>)

See “%SYSFUNC and %QSYSFUNC” on page 249

%QTRIM

Trims trailing blanks and returns a result that masks special characters and mnemonic operators

Type: Autocall macro

Requires: MAUTOSOURCE system option

Syntax

%QTRIM(*text* | *text expression*)

See “%TRIM and %QTRIM” on page 274

Note: Autocall macros are included in a library supplied by SAS Institute. This library may not be installed at your site or may 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 SAS Software Consultant. For more information, see Chapter 9 in *SAS Macro Language: Reference*. △

%QUOTE and %NRQUOTE

Mask special characters and mnemonic operators in a resolved value at macro execution

Type: Macro quoting functions

See also:

"%BQUOTE and %NRBQUOTE" on page 157
"%NRBQUOTE" on page 207
"%NRSTR" on page 207
"%SUPERQ" on page 226

Syntax

%QUOTE (*character string* | *text expression*)

%NRQUOTE (*character string* | *text expression*)

Details The %QUOTE and %NRQUOTE functions mask a character string or resolved value of a text expression during execution of a macro or macro language statement. They mask the following special characters and mnemonic operators:

```
+ – * / < > = ¬ ^ ~ ; , blank
AND OR NOT EQ NE LE LT GE GT
```

They also mask the following characters when they occur in pairs and when they are not matched and are marked by a preceding `%`:

```
' " ( )
```

In addition, %NRQUOTE masks

```
& %
```

%NRQUOTE is most useful when an argument may contain a macro variable reference or macro invocation that you do not want resolved.

For a description of quoting in SAS macro language, see Chapter 7 in *SAS Macro Language: Reference*.

Comparisons

- %QUOTE and %NRQUOTE mask 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, while %QUOTE and %NRQUOTE work when a macro executes.
- The %BQUOTE and %NRBQUOTE functions do not require that quotation marks and parentheses without a match be marked with a preceding `%`, while %QUOTE and %NRQUOTE do.
- %QUOTE and %NRQUOTE mask resolved values, while the %SUPERQ function prevents resolution of any macro invocations or macro variable references that may occur in a value.

Example

Example 1: Quoting a Value that May 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 may 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 implicitly 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

Converts a value to uppercase and returns a result that masks special characters and mnemonic operators

Type: Macro function

Syntax

%QUPCASE (*character string* | *text expression*)

See “%UPCASE and %QUPCASE” on page 276

RESOLVE

Resolves the value of a text expression during DATA step execution

Type: SAS function

Syntax

RESOLVE(*argument*)

argument
can be

- □ a text expression enclosed in single quotation marks (to prevent 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 *argument* references a nonexistent macro variable, RESOLVE returns the unresolved reference. These examples using text expressions show how to assign the text generated by macro LOCATE or assign the value of the macro variable NAME:

```
x=resolve('%locate');
x=resolve('&name');
```

- □ the name of a DATA step variable whose value is a text expression. For example, this example assigns the value of the text expression in the current value of the DATA step variable ADDR1 to X:

```
addr1='&locate';
x=resolve(addr1);
```

- □ a character expression that produces a text expression for resolution by the macro facility. For 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 explicitly 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 may 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

Example 1: 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;
```

Executing this program writes these lines to the SAS log:

```
VAR1=Holiday  VAR2=New Year  VAR3=New Year
NOTE: The data set WORK.TEST has 1 observations and 4 variables.
```

SASAUTOS=

Specifies one or more autocall libraries

Type: System option

Can be specified in:
Configuration file
OPTIONS window
OPTIONS statement
SAS invocation

Syntax

SASAUTOS= *library-specification* |
(*library-specification-1* . . . , *library-specification-n*)

library-specification
identifies a location that contains library members that contain a SAS macro definition. A location can be a SAS fileref or a host-specific location name enclosed in quotation marks. Each member contains a SAS macro definition.

(*library-specification-1* . . . , *library-specification-n*)
identifies two or more locations that contain library members that contain a SAS macro definition. A location can be a SAS fileref or a host-specific location name

enclosed in quotation marks. When you specify two or more autocall libraries, enclose the specifications in parentheses and separate them with either a comma or a blank space.

Details When SAS searches for an autocall macro definition, it opens and searches each location in the same order that it is specified in the SASAUTOS option. If SAS cannot open any specified location, it generates a warning message and sets the NOMAUTOSOURCE system option on. To use the autocall facility again in the same SAS session, you must specify the MAUTOSOURCE option again.

For more information, refer to Chapter 9 in *SAS Macro Language: Reference*.

Operating Environment Information: You specify a source library by using a fileref or by enclosing the host-specific location name in quotation marks. A valid library specification and its syntax are host specific. Although the syntax is generally consistent with the command-line syntax of your operating environment, it may include additional or alternate punctuation. For details, see the SAS documentation for your operating environment. △

SASMSTORE=

Specifies the libref of a SAS library with a catalog that contains, or will contain, stored compiled SAS macros

Type: System option

Can be specified in:
Configuration file
OPTIONS window
OPTIONS statement
SAS invocation

Syntax

SASMSTORE=*libref*

libref
specifies the libref of a SAS data library that contains, or will contain, a catalog of stored compiled SAS macros. This libref cannot be WORK.

%SCAN and %QSCAN

Search for a word that is specified by its position in a string

Type: Macro functions

See also:
"%NRBQUOTE" on page 207

"%STR and %NRSTR" on page 221

Syntax

%SCAN(*argument,n*<,*delimiters*>)

%QSCAN(*argument,n*<,*delimiters*>)

argument
is a character string or a text expression. If *argument* might contain a special character or mnemonic operator, listed below, use %QSCAN. If *argument* contains a comma, enclose *argument* in a quoting function, for example, %QUOTE(*argument*).

n
is 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.

delimiters
specifies an optional list of one or more characters that separate "words" or text expressions that yield one or more characters. To use a single blank or a single comma as the only delimiter, you must enclose the character in the %STR function, for example %STR( )or %STR(,). The delimiters recognized by %SCAN and %QSCAN vary between ASCII and EBCDIC systems. If you omit delimiters, SAS treats these characters as default delimiters:

ASCII systems
blank . < (+ & ! $ *); ^ – / , % |

EBCDIC systems
blank . < (+ | & ! $ *); ¬ – / , % | ¢

If *delimiters* includes any of the default delimiters for your system, the remaining default delimiters are treated as text.

To determine if you are using an ASCII or EBCDIC system, see the SAS companion for your operating system.

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
```

Comparisons

%QSCAN masks the same characters as the %NRBQUOTE function.

Example

Example 1: 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 this line:

```
X: %a*%b*%c
The third word in X, with SCAN: cccccc
The third word in X, with QSCAN: %c
```

SERROR

Controls whether the macro processor issues a warning message when a macro variable reference cannot be resolved

Type: System option

Can be specified in:
Configuration file
OPTIONS window
OPTIONS statement
SAS invocation

Default: SERROR

Syntax

SERROR | NOSERROR

SERROR
issues a warning message when the macro processor cannot match a macro variable reference to an existing macro variable.

NOSERROR
issues no warning messages when the macro processor cannot match a macro variable reference to an existing macro variable.

Details Several conditions can occur that prevent a macro variable reference from resolving. These conditions appear when

- 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.

%STR and %NRSTR

Mask special characters and mnemonic operators in constant text at macro compilation

Type: Macro quoting function

See also: "%NRQUOTE" on page 207

Syntax

%STR (*character-string*)

%NRSTR (*character-string*)

Details The %STR and %NRSTR functions mask a character string during compilation of a macro or macro language statement. They mask the following special characters and mnemonic operators:

```
+ – * / < > = ¬ ^ ~ ; , blank
AND OR NOT EQ NE LE LT GE GT
```

They also mask the following characters when they occur in pairs and when they are not matched and are marked by a preceding `%`:

```
' " ( )
```

In addition, %NRSTR also masks

```
& %
```

When an argument contains...	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.

Putting the same argument within nested %STR and %QUOTE functions is redundant. This example shows an argument that is masked at macro compilation by the %STR function and so remains masked at macro execution. Thus, in this example, the %QUOTE function used here has no effect.

```
%quote(%str(argument))
```

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 Chapter 7 in *SAS Macro Language: Reference*.

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 and %NRSTR mask the same items as %QUOTE and %NRQUOTE. However, %QUOTE and %NRQUOTE work during macro execution.
- If resolution of a macro expression will produce 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 allows the value of the macro variable TIME to contain leading blanks.

```
%let time=%str(   now);

%put Text followed by the value of time:&time;
```

Executing this example writes these lines 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)
```

Executing this program writes these lines to the SAS log:

```
The string contains 6 words.
```

Example 3: Quoting a Value That May 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);
```

Executing this program writes these lines to the SAS log:

```
%test&test - tset&sdrow owT
```

%SUBSTR and %QSUBSTR

Produce a substring of a character string

Type: Macro functions

See also: "%NRBQUOTE" on page 207

Syntax

%SUBSTR (*argument,position*<*,length*>)

%QSUBSTR (*argument,position*<*,length*>)

argument
is a character string or a text expression. If *argument* might contain a special character or mnemonic operator, listed below, use %QSUBSTR.

position
is 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 and %QSUBSTR issue a warning message and return a null value. An automatic call to %EVAL causes *n* to be treated as a numeric value.

length
is 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*, %SUBSTR and %QSUBSTR issue a warning message and return a substring containing the characters from *position* to the end of the string. By default, %SUBSTR and %QSUBSTR produce a string containing the characters from *position* to the end of the character string.

Details The %SUBSTR and %QSUBSTR functions produce a substring of *argument*, 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
```

Comparisons

%QSUBSTR masks the same characters as the %NRBQUOTE function.

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 statement

```
FILENAME HUMANRES "/dept/humanresource/report96";
```

Example 2: 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.));
```

Executing this program writes these lines 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.
```

Example 3: 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);
```

Executing these statements writes these lines to the SAS log:

```
C: &a &b
With SUBSTR: one
With QSUBSTR: &a
```

%SUPERQ

Masks all special characters and mnemonic operators at macro execution but prevents further resolution of the value

Type: Macro quoting function

See also:
"%NRBQUOTE" on page 207
"%BQUOTE and %NRBQUOTE" on page 157

Syntax

%SUPERQ (*argument*)

argument
is the name of a macro variable with no leading ampersand or a text expression that produces the name of a macro variable with no leading ampersand.

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
```

%SUPERQ is particularly useful for masking macro variables that might contain an ampersand or a percent sign when they are used with the %INPUT or %WINDOW statement, or the SYMPUT routine.

For a description of quoting in SAS macro language, see Chapter 7 in *SAS Macro Language: Reference*.

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*, while 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, while %NRBQUOTE attempts to resolve any macro references or macro variable values in the argument before masking the result.

Example

Example 1: 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;
```

Executing this program writes these lines 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. Because 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.

SYMBOLGEN

Controls whether the results of resolving macro variable references are displayed for debugging

Type: System option

Alias: SGEN | NOSGEN

Can be specified in:

Configuration file

OPTIONS window

OPTIONS statement

SAS invocation

Default: NOSYMBOLGEN

Syntax

SYMBOLGEN | NOSYMBOLGEN

SYMBOLGEN

displays the results of resolving macro variable references. This option is useful for debugging.

NOSYMBOLGEN

does not display results of resolving macro variable references.

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

Example 1: 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,data);
    title "%upcase(&proc) of %upcase(&data)";
%mend mktitle;

%macro runplot(ds);
   %if %sysprod(graph)=1 %then
      %do;
         %mktitle (gplot,&ds)
         proc gplot data=&ds;
            plot style*price
                / haxis=0 to 150000 by 50000;
         run;
         quit;
      %end;
   %else
      %do;
         %mktitle (plot,&ds)
         proc plot data=&ds;
            plot style*price;
         run;
         quit;
      %end;
%mend runplot;

%runplot(sasuser.houses)
```

Executing this program writes this SYMBOLGEN output to the SAS log:

```
SYMBOLGEN:  Macro variable DS resolves to sasuser.houses
SYMBOLGEN:  Macro variable PROC resolves to gplot
SYMBOLGEN:  Macro variable DATA resolves to sasuser.houses
SYMBOLGEN:  Macro variable DS resolves to sasuser.houses
```

SYMGET

Returns the value of a macro variable to the DATA step during DATA step execution

Type: SAS language function

See also:

"RESOLVE" on page 215
"SYMGETN" on page 231
"SYMPUT" on page 233
"SYMPUTN" on page 237

Syntax

SYMGET(*argument*)

argument
can be

- the name of a macro variable within quotation marks but without an ampersand. When a macro variable value contains another macro variable reference, SYMGET does not attempt to resolve the reference. If *argument* references a nonexistent macro variable, SYMGET returns a missing value. This example shows how to assign the value of the macro variable G to the DATA step variable X.

```
x=symget('g');
```

- the name of a DATA step character variable, specified with no quotation marks, which contains names of one or more macro variables. 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 illegal argument and sets the resulting value to missing. For 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 CODE supplies the name of a macro variable whose value is then assigned to KEY.

- a character expression that constructs a macro variable name. For example, this statement assigns the letter **s** and the number of the current iteration (using the automatic DATA step 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 illegal 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 may 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

Example 1: Retrieving Variable Values Previously Assigned from a Data Set

```
data dusty;
   input dept $ name $ salary @@;
   cards;
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 output shown in Output 13.1 on page 231.

Output 13.1 Intermediate Data Set and Final Report

```
                       Summary of Salary Information                     1
                        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


                       Salary Profiles for Employees                     2
                          Dusty Department Store


                                           Percent of  Percent of
                                           Department    Store
     OBS  Department  Employee  SALARY     Salary        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

In Screen Control Language (SCL) programs, returns the value of a global macro variable as a numeric value

Type: SCL function

See also:

"SYMGET" on page 228
"SYMPUT" on page 233
"SYMPUTN" on page 237

Syntax

SCL-variable=***SYMGETN***('*macro-variable*');

SCL variable
is the name of a numeric SCL variable to contain the value stored in *macro-variable*.

macro-variable
is the name of a global macro variable with no ampersand – note the single quotation marks. Or, the name of an SCL variable that contains the name of a global macro variable.

Details SYMGETN returns the value of a global macro variable as a numeric value and stores it in the specified numeric SCL variable. You can also use SYMGETN to retrieve the value of a macro variable whose name is stored in an SCL variable. For example, to retrieve the value of SCL variable UNITVAR, whose value is 'UNIT':

```
unitnum=symgetn(unitvar)
```

SYMGETN returns values when SCL programs execute. If SYMGETN cannot locate *macro-variable*, it returns a missing value.

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;
```

Note: It is inefficient to use SYMGETN to retrieve values that are not assigned with SYMPUTN and values that are not numeric. △

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.

Example

Example 1: Storing a Macro Variable Value as a Numeric Value In an SCL Program This statement stores the value of the macro variable UNIT in the SCL variable UNITNUM when the SCL program executes:

```
unitnum=symgetn('unit');
```

SYMPUT

Assigns a value produced in a DATA step to a macro variable

Type: SAS language routine

See also: “SYMGET” on page 228

Syntax

CALL SYMPUT(*macro-variable*, *value*);

macro-variable
can be

- a character string that is a SAS name, enclosed in quotes. For example, to assign the character string **testing** to macro variable NEW

```
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 assign them the values ANN, TOM, and BILL.

```
data team1;
   input position : $8. player : $12.;
   call symput(position,player);
cards;
shortstp Ann
pitcher Tom
frstbase Bill
;
```

- 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_ and assigns values to the macro variables POS1, POS2, and POS3.

```
data team2;
   input position : $12. player $12.;
   call symput('POS'||left(_n_), position);
   cards;
shortstp Ann
pitcher Tom
frstbase Bill
;
```

value
is the value to be assigned, which can be

- a string enclosed in quotes. 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 because 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.))));
cards;
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* does not exist, SYMPUT creates it. 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.

Concepts

Scope of Variables Created with SYMPUT SYMPUT puts the macro variable in the most local nonempty symbol table. In addition to a symbol table that contains a value, a symbol table is also considered nonempty if a computed %GOTO is present or the macro variable &SYSPBUFF is created at macro invocation time. (A computed %GOTO contains **`%`** or **`&`** and resolves to a label.)

If the local symbol table is empty and an executing macro contains a computed %GOTO and uses SYMPUT to create a macro variable, the variable is created in the local, empty symbol table and not in the nearest nonempty symbol table.

If the local symbol table is empty and an executing macro uses &SYSPBUFF and SYMPUT to create a macro variable, the macro is created in the local, empty symbol table and not in the nearest nonempty symbol table.

For more information on creating a variable with SYMPUT, see Chapter 5 in *SAS Macro Language: Reference*.

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, but 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 explicitly use 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;
```

Chapter 4, "Macro Processing" in *SAS Macro Language: Reference* provides details on 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));
   cards;
x
;
run;

%put char1 = ***&char1***;
%put char2 = ***&char2***;
```

Executing this program writes these lines 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***;
```

Executing this program writes these lines 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

Example 1: Creating Macro Variables and Assigning Them Values from a Data Set

```
data dusty;
   input dept $ name $ salary @@;
   cards;
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_;
```

Executing this program writes these lines this list of variables to the SAS log:

```
GLOBAL SCARPET 41000
GLOBAL SKITCHEN 46000
GLOBAL STV 48000
GLOBAL SGIFTS 27000
GLOBAL SBEDDING 59000
GLOBAL S_TOT 221000
```

SYMPUTN

In SCL programs, assigns a numeric value to a global macro variable

Type: SCL routine

See also:

"SYMGET" on page 228
"SYMGETN" on page 231
"SYMPUT" on page 233

Syntax

CALL SYMPUTN('*macro-variable*',*value*);

macro-variable
is the name of a global macro variable with no ampersand – note the single quotation marks. Or, 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

Example 1: 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);
```

SYSBUFFR

Contains text that is entered in response to a %INPUT statement when there is no corresponding macro variable

Type: Automatic macro variable (read and write)

Details Until the first execution of a %INPUT statement, SYSBUFFR has a null value. However, SYSBUFFR receives a new value during each execution of a %INPUT statement, either the text entered in response to the %INPUT statement where there is no corresponding macro variable or a null value. If a %INPUT statement contains no macro variable names, all characters entered are assigned to SYSBUFFR.

Example

Example 1: Assigning Text to SYSBUFFR This %INPUT statement accepts the values of the two macro variables–WATRFALL and RIVER:

```
%input watrfall river;
```

If you enter the following text, there is not a one-to-one match between the two variable names and the text:

```
Angel Tributary of Caroni
```

For example, you can submit these statements:

```
%put WATRFALL contains: *&watrfall*;
%put RIVER contains: *&river*;
%put SYSBUFFR contains: *&sysbuffr*;
```

After execution, they produce this output in the SAS log:

```
WATRFALL contains: *Angel*
RIVER contains: *Tributary*
SYSBUFFR contains: * of Caroni*
```

As the SAS log demonstrates, the text stored in SYSBUFFR includes leading and embedded blanks.

%SYSCALL

Invokes a SAS call routine

Type: Macro statement

Restriction: Allowed in macro definitions or in open code

See also: "%SYSFUNC and %QSYSFUNC" on page 249

Syntax

%SYSCALL *call-routine*<(*call-routine-argument(s)*)>;

call-routine
is a SAS System or user-written CALL routine created with SAS/TOOLKIT. 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 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.

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. △

Example

Example 1: Using the RANUNI Call Routine with %SYSCALL This example illustrates the %SYSCALL statement. The macro statement %SYSCALL RANUNI(A,B) invokes 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
```

SYSCC

Contains the current condition code that SAS returns to your operating environment (the operating environment condition code)

Type: Automatic macro variable (read and write)

Details SYSCC is a read/write automatic macro variable that enables you to reset the job condition code and to recover from conditions that prevent subsequent steps from running.

A normal exit internally to SAS is 0. The host code translates the internal value to a meaningful condition code by each host for each operating environment. &SYSCC of 0 at SAS termination is the value of success for that operating environment's return code.

Examples of successful condition codes are

Operating System	Value
OS/390	RC 0
Open/VMS	$STATUS = 1

The method to check the operating system return code is host dependent.

The warning condition code in SAS sets &SYSCC to 4.

SYSCHARWIDTH

Contains the character width value

Type: Automatic macro variable (read only)

Details The character width value is either 1 (narrow) or 2 (wide).

SYSCMD

Contains the last unrecognized command from the command line of a macro window

Restriction: Automatic macro variable (read and write)

Details

The value of SYSCMD is null before each execution of a %DISPLAY statement. If you enter a word or phrase on the command line of a macro window and the windowing

environment does not recognize the command, SYSCMD receives that word or phrase as its value. This is the only way to change the value of SYSCMD, which otherwise is a read-only variable. Use SYSCMD to enter values on the command line that work like user-created windowing commands.

Example

Example 1: Processing Commands Entered In a Macro Window The macro definition START creates a window in which you can use the command line to enter any windowing command. If you type an invalid command, a message informs you that the command is not recognized. When you type QUIT on the command line, the window closes and the macro terminates.

```
%macro start;
  %window start
     #5  @28 'Welcome to the SAS System'
     #10  @28 'Type QUIT to exit';
  %let exit = 0;
  %do %until (&exit=1);
     %display start;
     %if &syscmd ne %then %do;
        %if %upcase(&syscmd)=QUIT %then %let exit=1;
        %else %let sysmsg=&syscmd not recognized;
     %end;
  %end;
%mend start;
```

SYSDATE

Contains the date that a SAS job or session began executing

Restriction: Automatic macro variable (read only)

See also: "SYSDATE9" on page 242

Details SYSDATE contains a SAS date value in the DATE7. format, which displays a two-digit date, the first three letters of the month name, and a two-digit year. The date does not change during the individual job or session. As an 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

Example 1: Formatting a SYSDATE Value Macro FDATE assigns a format 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.)
title "Tests for &fdate";
```

If you execute this macro on July 28, 1998, SAS sees the statements:

```
DATA _NULL_;
   CALL SYMPUT("FDATE",LEFT(PUT("28JUL98"D,WORDDATE.)));
RUN;
TITLE "Tests for July 28, 1998";
```

For another method of formatting the current date, see the %SYSFUNC and %QSYSFUNC functions.

SYSDATE9

Contains the date that a SAS job or session began executing

Restriction: Automatic macro variable (read only)

See also: "SYSDATE" on page 241

Details SYSDATE9 contains a SAS date value in the DATE9. format, which displays a two-digit date, the first three letters of the month name, and a four–digit year. The date does not change during the individual job or session. As an 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

Example 1: Formatting a SYSDATE9 Value Macro FDATE assigns a format 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.)
title "Tests for &fdate";
```

If you execute this macro on July 28, 1998, SAS sees the statements:

```
DATA _NULL_;
   CALL SYMPUT("FDATE",LEFT(PUT("28JUL1998"D,WORDDATE.)));
RUN;
TITLE "Tests for July 28, 1998";
```

For another method of formatting the current date, see the %SYSFUNC and %QSYSFUNC functions.

SYSDAY

Contains the day of the week that 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 SAS session today.

Example

Example 1: Identifying the Day When a SAS Session Started The following statement identifies the day and date when a SAS session started running.

```
%put This SAS session started running on: &sysday, &sysdate.;
```

Executing this statement on Thursday, December 18, 1997 for a SAS session that began executing on Tuesday, December 16, 1997, writes this to the SAS log:

```
This SAS session started running on: Tuesday, 16DEC97
```

SYSDEVIC

Contains the name of the current graphics device

Type: Automatic macro variable (read and write)

Details The current graphics device is the one specified at invocation of SAS. You can specify the graphics device on the command line in response to a prompt when you use a product that uses SAS/GRAPH. You can also specify the graphics device in a configuration file. The name of the current graphics device is also the value of the SAS system option DEVICE=.

For details, see the SAS documentation for your operating environment.

Comparisons

Assigning a value to SYSDEVIC is the same as specifying a value for the DEVICE= system option.

SYSDMG

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:

Value	Description
0	No repair of damaged data sets in this session. (Default)
1	One or more automatic repairs of damaged data sets has occurred.
2	One or more user-requested repairs of damaged data sets has 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.

SYSDSN

Contains the libref and name of the most recently created SAS data set

Type: Automatic macro variable (read and write)

See also: “SYSLAST” on page 254

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 eight blanks followed by _NULL_ followed by two more blanks.

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

Example 1: Comparing Values Produced by SYSDSN and SYSLAST Create a data set WORK.TEST and then enter the following statements:

```
%put Sysdsn produces:  *&sysdsn*;
%put Syslast produces: *&syslast*;
```

Executing these statements writes to the SAS log:

```
Sysdsn produces:  *WORK    TEST    *
Syslast produces: *WORK.TEST    *
```

When the libref or data set name contain fewer than eight characters, SYSDSN maintains the blanks for the unused characters. SYSDSN does not display a period between the libref and data set name fields.

SYSENV

Reports whether SAS is running interactively

Type: Automatic macro variable (read only)

Details The value of SYSENV is independent of the source of input. Values for SYSENV are

FORE
: when the SAS system option TERMINAL is in effect. For example, the value is FORE when you run SAS interactively through a windowing environment.

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, or when you invoke the SAS System with the name of a file that contains SAS code.

You can use SYSENV to check the execution mode before submitting code that requires interactive processing. To use a %INPUT statement, the value of SYSENV must be FORE. For details, 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 details, see the SAS documentation for your operating environment. △

SYSERR

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 can contain:

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.

Value	Description
4	Execution completed successfully but with warning messages.
>4	An error occurred. The value returned is procedure dependent.

%SYSEVALF

Evaluates arithmetic and logical expressions using floating-point arithmetic

Type: Macro function

See also: "%EVAL" on page 171

Syntax

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

expression
is an arithmetic or logical expression to evaluate

conversion-type
optionally 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

BOOLEAN
returns

0 if the result of the expression is 0 or missing

1 if the result is any other value.

For 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. For example,

```
%sysevalf(1 + 1.1,ceil)       /* returns  3 */
%sysevalf(-1 -2.4,ceil)       /* returns -3 */
%sysevalf(-1 + 1.e-11,ceil)   /* returns -1 */
%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 an missing value produces a missing value. For example,

```
%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 an missing value produces a missing value. For example,

```
%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 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. Specifying a conversion type can prevent problems when %SYSEVALF returns missing or floating point values to macro expressions or macro variables that are used in other macro expressions that require an integer value.

For details on evaluation of expressions by the SAS macro language, see Chapter 6 in *SAS Macro Language: Reference*.

Comparisons

- %SYSEVALF supports floating point numbers. However, %EVAL performs only integer arithmetic.
- You must explicitly 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.

Example

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)
```

Executing this program writes these lines 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
```

%SYSEXEC

Issues operating environment commands

Type: Macro statement

Restriction: Allowed in macro definitions or open code

See also:

"SYSSCP and SYSSCPL" on page 269
"SYSRC" on page 266

Syntax

%SYSEXEC< *command*>;

no argument
puts you into operating environment mode under most operating environments, where you can issue operating environment commands and return to your SAS session.

command
is any operating environment command. If *command* may contain a semicolon, use a macro quoting function.

Details The %SYSEXEC statement causes the operating environment to immediately execute the command you specify and assigns any return code from the operating environment to the automatic macro variable SYSRC. Use the %SYSEXEC statement and the automatic macro variables SYSSCP and SYSSCPL to write portable macros that run under multiple operating environments. (See "Example.")

Operating Environment Information: These items related to the use of the %SYSEXEC statement are operating environment specific:

- the availability of the %SYSEXEC statement in batch processing, noninteractive mode, or interactive line mode
- the way you return from operating environment mode to your SAS session after executing the %SYSEXEC statement with no argument
- the commands to use with the %SYSEXEC statement
- the return codes you get in the automatic macro variable SYSRC.

For details, see the SAS documentation for your operating environment. △

Comparisons

The %SYSEXEC statement is analogous to the X statement and the X windowing environment command. However, unlike the X statement and the X windowing

environment command, host commands invoked with %SYSEXEC should not be enclosed in quotation marks.

Example

Example 1: Executing Operating Environment-Specific Utility Programs In this macro, ACLIB, the %SYSEXEC statement executes one of two operating environment utility programs based on the value of the automatic macro variable SYSSCP. If the value of SYSSCP is anything other than **OS** or **VMS**, ACLIB writes a message in the SAS log indicating that no utilities are available.

```
%macro aclib;
       %if %upcase(&sysscp)=OS %then
           %sysexec ex 'dept.tools.clist(tiefiles)';
       %else %if %upcase(&sysscp)=VMS %then
           %sysexec @tiefiles;
       %else %put NO UTILITIES AVAILABLE ON &sysscp..;
    %mend aclib;
```

SYSFILRC

Contains the return code from the last FILENAME statement

Type: Automatic macro variable (read and write)

Details The return code reports whether the last FILENAME statement executed correctly. 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.
Values for SYSFILRC are

Value	Description
0	The last FILENAME statement executed correctly.
≠0	The last FILENAME statement did not execute correctly.

%SYSFUNC and %QSYSFUNC

Execute SAS functions or user-written functions

Type: Macro functions

Syntax

%SYSFUNC (*function*(*argument(s)*)<,*format*>)

%QSYSFUNC (*function*(*argument(s)*)<,*format*>)

function

is the name of the function to execute. This function can be a SAS function or a function written with SAS/TOOLKIT software. The function cannot be a macro function.

All SAS functions, except those listed in the table Table 13.1 on page 251, can be used with %SYSFUNC and %QSYSFUNC.

You cannot nest functions to be used with a single %SYSFUNC. However, you can nest %SYSFUNC calls, for example,

```
%let x=%sysfunc(trim(%sysfunc(left(&num))));
```

Appendix 3, “Syntax for the Selected SAS Functions Used with %SYSFUNC,” in *SAS Macro Language: Reference*, shows the syntax of SAS functions introduced with Release 6.12 used with %SYSFUNC.

argument(s)

is 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. If *argument* might contain a special character or mnemonic operator, listed below, use %QSYSFUNC.

format

is an optional format to apply to the result of *function*. This format can be provided by the SAS System, generated by PROC FORMAT, or created with SAS/TOOLKIT. By default, numeric results are converted to a character string using the BEST12. format and character results are used as they are, without formatting or translation.

Details 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, but do not require quotation marks when used within %SYSFUNC. These statements show the difference:

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

%SYSFUNC does not mask special characters or mnemonic operators in its result. %QSYSFUNC masks the following special characters and mnemonic operators in its result:

```
& % ' " ( ) + – * / < > = ¬ ^ ~ ; , blank
AND OR NOT EQ NE LE LT GE GT
```

When a function called by %SYSFUNC or %QSYSFUNC requires a numeric argument, the macro facility converts the argument to a numeric value. %SYSFUNC and %QSYSFUNC can return a floating point number when the function they execute supports floating point numbers.

Table 13.1 SAS Functions Not Available with %SYSFUNC and %QSYSFUNC

DIF	DIM	HBOUND
IORCMSG	INPUT	LAG
LBOUND	MISSING	PUT
RESOLVE	SYMGET	All Variable Information Functions

Note: Intead of INPUT and PUT, which are not available with %SYSFUNC and %QSYSFUNC, use INPUTN, INPUTC, PUTN, and PUTC in Screen Control Language. △

Note: The Variable Information functions include functions such as VNAME and VLABEL. For a complete list, see "Functions and CALL Routines" in *SAS Language Reference: Dictionary*. △

CAUTION:
Values returned by SAS functions may 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

%QSYSFUNC masks the same characters as the %NRBQUOTE 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";
```

Executing this statement on July 18, 2000, produces this TITLE statement:

```
title "July 18, 2000 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)
```

Executing this program writes these lines 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;
```

Executing these statements writes these lines 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)
```

Executing this program produces the statements:

```
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 utilized 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,nvarsp,nobsp);
   %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));
      %end;
   %else
      %put Open for data set &dset failed - %sysfunc(sysmsg());
%mend obsnvars;

%obsnvars(sasuser.houses,nvars,nobs)

%put &dset has &nvars variable(s) and &nobs observation(s).;
```

Executing this program writes these lines to the SAS log:

```
sasuser.houses has 6 variable(s) and 15 observation(s).
```

%SYSGET

Returns the value of the specified operating environment variable

Type: Macro function

Syntax

%SYSGET(*environment-variable*)

environment-variable
is the name of an environment variable. The case of *environment-variable* must agree with the case that is stored on the operating environment.

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.

For details, see the SAS documentation for your operating environment.

Example

Example 1: Using SYSGET in a UNIX Operating Environment This example returns the id of a user on a UNIX operating environment:

```
%let person=%sysget(USER);
%put User is &person;
```

Executing these statements for user ABCDEF prints this in the SAS log:

```
User is abcdef
```

SYSINDEX

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

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

Contains the name of the current batch job or userid

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

Contains the name of the SAS data file created most recently

Type: Automatic macro variable (read and write)

See also: "SYSDSN" 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. If no SAS data set has been created in the current program, the value of SYSLAST is _NULL_, with no leading or trailing blanks.

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.

Examples

Example 1: Comparing Values Produced by SYSLAST and SYSDSN Create the data set FIRSTLIB.SALESRPT and then enter the following statments:

```
%put Sysdsn produces:  *&sysdsn*;
%put Syslast produces: *&syslast*;
```

Executing these statements writes this 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.

SYSLCKRC

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 lock data objects in data libraries accessed through SAS/SHARE software. Values for SYSLCKRC are

Value	Description
0	The last LOCK statement executed correctly.
≠0	The last LOCK statement did not execute correctly.

For more information, see the documentation for SAS/SHARE software.

SYSLIBRC

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 data library referenced by the last LIBNAME statement exists. As an example, you could use SYSLIBRC to confirm that a libref is allocated before you attempt to access a permanent data set.

Values for SYSLIBRC are

Value	Description
0	The last LIBNAME statement executed correctly.
≠0	The last LIBNAME statement did not execute correctly.

%SYSLPUT

Creates a new macro variable or modifies the value of an existing macro variable on a remote host or server

Type: Macro Statement

Requires: SAS/CONNECT

Restriction: Allowed in macro definitions or open code

See also:

“%LET” on page 190
“%SYSRPUT” on page 267

Syntax

%SYSLPUT *macro-variable*=<*value*>;

macro-variable
is either the name of a macro variable or a macro expression that produces a macro variable name. The name can refer to a new or existing macro variable on a remote host or server.

value
is a string or a macro expression that yields a string. 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.

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 SAS session or client and a remote SAS session or server using the SIGNON command or SIGNON statement. For more information, see the documentation for SAS/CONNECT software.

SYSMENV

Contains the invocation status of the macro that is currently executing

Type: Automatic macro variable (read only)

Details Values for SYSMENV are

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.

SYSMSG

Contains text to display in the message area of a macro window

Type: Automatic macro variable (read and write)

Details Values assigned to SYSMSG do not require quotation marks. The value of SYSMSG is set to null after each execution of a %DISPLAY statement.

Example

Example 1: Using SYSMSG This example shows that text assigned to SYSMSG is cleared after the %DISPLAY statement.

```
%let sysmsg=Press ENTER to continue.;
%window start
   #5  @28 'Welcome to the SAS System';
%display start;

%put Sysmsg is: *&sysmsg*;
```

Executing this program writes this to the SAS log:

```
Sysmsg is: **
```

SYSPARM

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 details, see the SAS documentation for your operating environment.

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

Example 1: Passing a Value to a Procedure In this example, you invoke SAS on a UNIX operating environment on September 20, 2001 (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.resources.priority.rept;
title "%sysfunc(date(),worddate.)";
title2;
title3 'Active Projects By Priority';
run;
```

SAS sees the following:

```
proc report data=dept.projects
     report=test.resources.priority.rept;
title "September 20, 2001";
title2;
title3 'Active Projects By Priority';
run;
```

SYSPARM=

Specifies a character string that can be passed to SAS programs

Type: System option

Can be specified in:
Configuration file
OPTIONS window
OPTIONS statement
SAS invocation

Syntax

SYSPARM='*character-string*'

character-string
is a character string, enclosed in quotation marks, with a maximum length of 200.

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 details, see the SAS documentation for your operating environment. △

Example

Example 1: Passing a User Identification to a Program This example uses the SYSPARM option to pass a user identification to a program.

```
options sysparm='usr1';

data a;
   length z $100;
   if sysparm()='usr1' then z="&sysparm";
run;
```

SYSPBUFF

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.

Example

Example 1: 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 dsname=%scan(&syspbuff,&num);
   %do %while(&dsname ne);
      proc print data=&dsname;
      run;
      %let num=%eval(&num+1);
      %let dsname=%scan(&syspbuff,&num);
   %end;
%mend printz;

%printz(purple,red,blue,teal)
```

Executing this program writes this line to the SAS log:

```
Syspbuff contains: (purple,red,blue,teal)
```

SYSPROCESSID

Contains the process id of the current SAS process

Type: Automatic macro variable (read only)

Default: null

Details The process id is a 32–character hexadecimal string. The default value is null.

Example

Example 1: 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
```

SYSPROCESSNAME

Contains the process name of the current SAS process

Type: Automatic macro variable (read only)

Example

Example 1: Using SYSPROCESSNAME to Display the Current SAS Process Name The following statement writes the name of the current SAS process to the log:

```
%put &sysprocessname;
```

If you submit this statement in the SAS windowing environment of your second SAS session, the following line is written to the SAS log:

```
DMS Process (2)
```

%SYSPROD

Reports whether a SAS software product is licensed at the site

Type: Macro function

See also:

"%SYSEXEC" on page 248
"SYSSCP and SYSSCPL" on page 269
"SYSVER" on page 273

Syntax

%SYSPROD (*product*)

product

can be a character string or text expression that yields a code for a SAS product. Commonly used codes are

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, see your SAS site representative.

Details %SYSPROD can return

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

Example 1: Verifying SAS/GRAPH Installation Before Running the GPLOT Procedure This example uses %SYSPROD to determine whether to execute a PROC GPLOT statement or a PROC PLOT statement, based on whether SAS/GRAPH software has been installed.

```
%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 "PLOT of %upcase(&ds)";
         proc plot data=&ds;
            plot style*price;
         run;
         quit;
      %end;
%mend runplot;

%runplot(sasuser.houses)
```

Executing this program when SAS/GRAPH is installed, generates the following statements:

```
TITLE "GPLOT of SASUSER.HOUSES";
PROC GPLOT DATA=SASUSER.HOUSES;
PLOT STYLE*PRICE / HAXIS=0 TO 150000 BY 50000;
RUN;
```

%SYSRC

Returns a value corresponding to an error condition

Type: Autocall macro

Requires: MAUTOSOURCE system option

Syntax

%SYSRC(*character-string*)

character-string
is one of the mnemonic values listed in Table 13.2 on page 263 or a text expression that produces the mnemonic value.

Note: Autocall macros are included in a library supplied by SAS Institute. This library may not be installed at your site or may 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 SAS Software Consultant. For more information, see Chapter 9 in *SAS Macro Language: Reference*. △

Details 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 system 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.

Table 13.2 on page 263 lists the mnemonic values to specify with the SYSRC function and a description of the corresponding error.

Table 13.2 Mnemonics for Warning and Error Conditions

Mnemonic	Description
Library Assign/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.

Mnemonic	Description
_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.
_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 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.
SAS File Open and Update Messages	
_SEBDSNM	The file name is not a valid SAS name.
_SEDLREC	The record has been deleted from the file.

Mnemonic	Description
_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.
_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.

Mnemonic	Description
_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.
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.

Comparison

The SYSRC autocall macro and the SYSRC automatic macro variable are not the same. For more information, see “SYSRC” on page 266.

Example

Example 1: Examining the Value of _IORC_ The following DATA step illustrates using the autocall macro SYSRC and the automatic variable _IORC_ to control writing a message to the SAS log:

```
data big;
   modify big trans;
   by id;
   if _iorc_=%sysrc(_dsenmr) then put 'WARNING: Check ID=' id;
run;
```

SYSRC

Contains the last return code generated by your operating system

Type: Automatic macro variable (read and write)

Details The code returned by SYSRC is based on commands you execute using the X statement in open code, the X command in a windowing environment, or the %SYSEXEC, %TSO, or %CMS macro statements. Return codes are integers. The default value of SYSRC is 0.

You can use SYSRC to check the return code of a system command before you continue with a job. For return code examples, see the SAS companion for your operating environment.

%SYSRPUT

Assigns the value of a macro variable on a remote host to a macro variable on the local host

Type: Macro statement

Requires: SAS/CONNECT

Restriction: Allowed in macro defintions or open code

See also:

"SYSERR" on page 245
"SYSINFO" on page 254
"%SYSLPUT" on page 256

Syntax

%SYSRPUT *local-macro-variable*=*remote-macro-variable*;

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 stored on the local host.

remote-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 stored on a remote host.

Details The %SYSRPUT 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 referencing environment of the local host.

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

The %SYSRPUT statement is useful for capturing the value of the automatic macro variable SYSINFO and passing that value to the local host. SYSINFO contains return-code information provided by some SAS procedures. Both the UPLOAD and the DOWNLOAD procedures of SAS/CONNECT can update the macro variable SYSINFO and set it to a nonzero value when the procedure terminates due to errors. You can use %SYSRPUT on the remote host to send the value of the SYSINFO macro variable back to the local SAS session. Thus, you can submit a job to the remote host and test whether a PROC UPLOAD or DOWNLOAD step has successfully completed before beginning another step on either the remote host or the local host.

For details on 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

Example 1: 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;
```

SYSSCP and SYSSCPL

Contain an identifier for your operating environment

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.

Table 13.3 on page 269 lists the values for SYSSCP and SYSSCPL.

Table 13.3 SYSSCP and SYSSCPL Values

Platform	SYSSCP Value	SYSSCPL Value
386 BCS	386 BCS	
AIX	RS6000	AIX
AIX/ESA	AIX_370	
AIX/PS2	AIX_370	
ALPHA/OSF	ALXOSF	
ALPHA/VMS	VMS_AXP	
CONVEX	CONVEX	
DigitalUnix	ALXOSF	DEC OSFI
DGUX	DG UX	
HPU3	HP 300	
HPUX	HP 800	HP-UX
IABI	386 ABI	386 ABI
LINUX	LINUX	
MAC68000	MAC	MAC_M68
MAC PowerPC	MAC	MAC_MPP
MIPS	UMIPS	
MIPS ABI	MIPS ABI	
MS-DOS		
MVS	OS	MVS
NEXT	NEXT	
OS/2	OS2	
PC-DOS	PC DOS	
PRIMOS	PRIMOS	
RS6000	RS6000	
SEQUENT IAB	SEQUENT	

Platform	SYSSCP Value	SYSSCPL Value
SGI MAX	IRIX	
SIEMENS	SINIX	
SOLARIS2	SUN 4	Solaris
SR10	SR10	
SUN3	SUN 3	
SUN4.1.x	SUN 4	SunOS
SUNOS	SUN 386i	
ULTRIX	ULTRIX	
VAX	VMS	
VM/CMS	CMS	VM_ESA
VMS	VMS	
VSE	VSE	
WINDOWS	WIN	
WINDOWS 32S	WIN	WIN_32S
WINDOWS 95	WIN	WIN_95
WINDOWS/NT	WIN	WIN_NT
NT Server	WIN	WIN_NTSV

Example

Example 1: Deleting a Temporary File on a Platform Running SAS The macro DELFILE locates the platform that is running SAS and deletes the TMP file. FILEREF is a global macro variable that contains the fileref for the TMP file.

```
%macro delfile;
   %if  /* HP Unix */&sysscp=HP 800 or &sysscp=HP 300
   %then
      %do;
         X "rm &fileref..TMP";
      %end;
   %else %if            /* VMS */&sysscp=VMS
   %then
      %do;
          X "DELETE &fileref..TMP;*";
      %end;
   %else %if  /* DOS-LIKE PLATFORMS */&sysscp=OS2 or &sysscp=WIN
   %then
      %do;
          X "DEL &fileref..TMP";
      %end;
   %else %if  /* CMS */&sysscp=CMS
   %then
      %do;
         X "ERASE &fileref TEMP A";
      %end;
%mend delfile;
```

SYSSCPL

Contains the name of your operating environment

Automatic macro variable (read only)

See "SYSSCP and SYSSCPL" on page 269.

SYSSITE

Contains the number assigned to your site

Type: Automatic macro variable (read only)

Details SAS Institute assigns a site number to each site that licenses SAS software. The number displays in the SAS log.

SYSSTARTID

Contains the id generated from the last STARTSAS statement

Type: Automatic macro variable (read only)
Default: null

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

Example 1: Using SYSSTARTID to Display the SAS Process ID from the Most Recent STARTSAS Statement 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

Contains the process name generated from the last STARTSAS statement

Type: Automatic macro variable (read only)

Default: null

Example

Example 1: Using SYSSTARTNAME to Display the SAS Process Name from the Most Recent STARTSAS Statement 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)
```

SYSTIME

Contains the time 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

Example 1: Using SYSTIME to Display the Time that a SAS Session Started The following statement displays the time a SAS session started.

```
%put This SAS session started running at: &systime;
```

Executing this statement at 3 p.m. when your SAS session began executing at 9:30 a.m. writes to the SAS log:

```
This SAS session started running at: 09:30
```

SYSUSERID

Contains the userid or login of the current SAS process

Type: Automatic macro variable (read only)

Example

Example 1: Using SYSUSERID to Display the Userid for the Current SAS Process The following code, when submitted from the current SAS process, writes the userid or login for the current SAS process to the SAS log:

```
%put &sysuserid;
```

A userid, such as the following, is written to the SAS log:

```
MyUserid
```

SYSVER

Contains the release number of SAS software that is running

Type: Automatic macro variable (read only)

See also: "SYSVLONG" on page 273

Comparison

SYSVER provides the release number of the SAS software that is running. You can use SYSVER to check for the release of the SAS System before running a job with newer features.

Example

Example 1: 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 (for a user of Release 6.12) writes this to the SAS log:

```
I am using release: 6.12
```

SYSVLONG

Contains the release number and maintenance level of SAS software that is running

Type: Automatic macro variable (read only)

See also: "SYSVER" on page 273

Comparisons

SYSVLONG provides the release number and maintenance level of SAS software, in addition to the release number.

Example

Example 1: Identifying a SAS Maintenance Release The following statement displays information identifying the SAS release being used.

```
%put I am using maintenance release: &sysvlong;
```

Submitting this statement (for a user of Release 6.12) writes this to the SAS log:

```
I am using maintenance release: 6.12.0005P123199
```

%TRIM and %QTRIM

Trim trailing blanks

Type: Autocall macro

Requires: MAUTOSOURCE system option

Syntax

%TRIM(*text* | *text expression*)

%QTRIM(*text* | *text expression*)

Note: Autocall macros are included in a library supplied by SAS Institute. This library may not be installed at your site or may 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 SAS Software Consultant. For more information, see Chapter 9 in *SAS Macro Language: Reference*. △

Details The TRIM macro and the QTRIM macro both trim trailing blanks. If the argument might contain a special character or mnemonic operator, listed below, use %QTRIM.

QTRIM 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:

```
& % ' " ( ) + – * / < > = ¬ˆ ~ ; , blank
AND OR NOT EQ NE LE LT GE GT
```

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 %upcase(&dsn).;
%mend numobs;
```

```
%numobs(sample)
```

Invoking the NUMOBS macro generates the following statements:

```
DATA _NULL_;
SET SAMPLE NOBS=COUNT;
CALL SYMPUT('num', LEFT(PUT(COUNT,8.)));
STOP;
RUN;
```

If the data set SAMPLE contains six observations, then the %PUT statement writes this line to the SAS log:

```
There were 6 observations in SAMPLE.
```

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*
```

%UNQUOTE

During macro execution, unmasks all special characters and mnemonic operators for a value

Type: Macro function

See also:
"%BQUOTE and %NRBQUOTE" on page 157
"%NRBQUOTE" on page 207
"%NRQUOTE" on page 207
"%NRSTR" on page 207
"%QUOTE and %NRQUOTE" on page 213
"%STR and %NRSTR" on page 221
"%SUPERQ" on page 226

Syntax

%UNQUOTE (*character string* | *text expression*)

Details The %UNQUOTE function unmasks a value so that special characters that it may 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 Chapter 7 in *SAS Macro Language: Reference*.

Example

Example 1: 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 may 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 explicitly unmasks the value of TESTVAL.

```
%let val = aaa;
%let testval = %str(%'&val%');

data _null_;
  val = %unquote(&testval);
  put 'VAL =' val;
run;
```

This program prints this to the SAS log:

```
VAL=aaa
```

%UPCASE and %QUPCASE

Convert values to uppercase

Type: Macro functions

See also:

"%LOWCASE and %QLOWCASE" on page 192

"%NRBQUOTE" on page 207

"%QLOWCASE" on page 212

Syntax

%UPCASE (*character string* | *text expression*)

%QUPCASE(*character string* | *text expression*)

Details The %UPCASE and %QUPCASE functions convert 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 a value might contain a special character or mnemonic operator, use %QUPCASE.

If the argument might contain a special character or mnemonic operator, listed below, use %QUPCASE. %QUPCASE masks the following special characters and mnemonic operators in its result:

```
& % ' " ( ) + – * / < > = ¬ ^ ~ ; , blank
AND OR NOT EQ NE LE LT GE GT
```

%UPCASE and %QUPCASE are useful in the comparison of values because the macro facility does not automatically convert lowercase characters to uppercase before comparing values.

Comparison

- %QUPCASE masks the same characters as the %NRBQUOTE function.
- To convert characters to lowercase, use the %LOWCASE or %QLOWCASE autocall macro.

Examples

Example 1: Capitalizing a Value to be Compared In this example, the macro RUNREPT compares a value input for the macro variable MONTH to the string DEC. If the uppercase value of the response is DEC, then PROC FSVIEW runs on the data set REPORTS.ENDYEAR. Otherwise, PROC FSVIEW runs on the data set with the name of the month in the REPORTS data library.

```
%macro runrept(month);
   %if %upcase(&month)=DEC %then
       %str(proc fsview data=reports.endyear; run;);
   %else %str(proc fsview data=reports.&month; run;);
%mend runrept;
```

You can invoke the macro in any of these ways to satisfy the %IF condition:

```
%runreport(DEC)
%runreport(Dec)
%runreport(dec)
```

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);
```

Executing these statements writes this to the SAS log:

```
UPCASE produces: begin
QUPCASE produces: &A
```

%VERIFY

Returns the position of the first character unique to an expression

Type: Autocall macro

Requires: MAUTOSOURCE system option

Syntax

%VERIFY(*source*, *excerpt*)

source
is text or a text expression. This is the text 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*.

Note: Autocall macros are included in a library supplied by SAS Institute. This library may not be installed at your site or may 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 SAS Software Consultant. For more information, see Chapter 9 in *SAS Macro Language: Reference*. △

Details %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

Example 1: Testing for a Valid Fileref The ISNAME macro checks a string to see if 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;

%isname(file1)
%isname(1file)
%isname(filename1)
%isname(file$)
```

Executing this program writes this 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$: The fileref cannot contain "$".
```

%WINDOW

Defines customized windows

Type: Macro Statement

Restriction: Allowed in macro definitions or open code

See also:
“%DISPLAY” on page 164
“%INPUT” on page 182
“%KEYDEF” on page 185

Syntax

%WINDOW *window-name*< *window-option(s)*>*group-definition(s)* | *field-definition(s)*;

window-name
names the window. *Window-name* must be a SAS name.

window-option(s)
specifies the characteristics of the window as a whole. Specify all window options before any field or group definitions. These window options are available:

COLOR=*color*
specifies the color of the window background. The default color of the window and the contents of its fields are both device-dependent. *Color* can be one of these:

BLACK

BLUE

BROWN

CYAN

GRAY (or GREY)

GREEN

MAGENTA

ORANGE

PINK

RED

WHITE

YELLOW

Operating Environment Information: The representation of colors may vary, depending on the display device you use. In addition, on some display devices the background color affects the entire window; on other display devices, it affects only the window border. △

COLUMNS=*columns*

specifies the number of display columns in the window, including borders. A window can contain any number of columns and can extend beyond the border of the display, which is useful when you need to display a window on a device larger than the one on which you developed it. By default, the window fills all remaining columns in the display.

Operating Environment Information: The number of columns available depends on the type of display device you use. Also, the left and right borders each use from 0 to 3 columns on the display depending on your display device. If you create windows for display on different types of display devices, make sure all fields can be displayed in the narrowest window. △

ICOLUMN=*column*

specifies the initial column within the display at which the window is displayed. By default, the macro processor begins the window at column 1 of the display.

IROW=*row*

specifies the initial row (line) within the display at which the window is displayed. By default, the macro processor begins the window at row 1 of the display.

KEYS=<<*libref.* >*catalog.* >*keys-entry*

specifies the name of a KEYS catalog entry that contains the function key definitions for the window. If you omit *libref* and *catalog*, SAS uses SASUSER.PROFILE.*keys-entry*.

If you omit the KEYS= option, SAS uses the current function key settings defined in the KEYS window.

MENU=<<*libref.* >*catalog.*>*pmenu-entry*

specifies the name of a menu you have built with the PMENU procedure. If you omit *libref* and *catalog*, SAS uses SASUSER.PROFILE.*pmenu-entry*.

ROWS=*rows*

specifies the number of rows in the window, including borders. A window can contain any number of rows and can extend beyond the border of the display, which is useful when you need to display a window on a device larger than the one on which you developed it. If you omit a number, the window fills all remaining rows in the display.

Operating Environment Information: The number of rows available depends on the type of display device you use. △

group-definition

names a group and defines all fields within a group. The form of *group definition* is

GROUP=*group field-definition* <. . . *field-definition-n*>

where *group* names a group of fields that you want to display in the window collectively. A window can contain any number of groups of fields; if you omit the GROUP= option, the window contains one unnamed group of fields. *Group* must be a SAS name.

Organizing fields into groups allows you to create a single window with several possible contents. To refer to a particular group, use *window.group*.

field-definition

identifies and describes a macro variable or string you want to display in the window. A window can contain any number of fields.

You use a field to identify a macro variable value (or constant text) to be displayed, its position within the window, and its attributes. Enclose constant text in quotation marks. The position of a field is determined by beginning row and column. The attributes that you can specify include color, whether you can enter a value into the field, and characteristics such as highlighting.

The form of a field definition containing a macro variable is

<*row*> <*column*> *macro-variable*<*field-length*> <*options*>

The form of a field definition containing constant text is

<*row*> <*column*>*'text'* | *"text"*<*options*>

The elements of a field definition are

row

specifies the row (line) on which the macro variable or constant text is displayed. Each row specification consists of a pointer control and, usually, a macro expression that generates a number. These row pointer controls are available:

#*macro-expression*

specifies the row within the window given by the value of the macro expression. The macro expression must either be a positive integer or generate a positive integer.

/ (*forward slash*)

moves the pointer to column 1 of the next line.

The macro processor evaluates the macro expression when it defines the window, not when it displays the window. Thus, the row position of a field is fixed when the field is being displayed.

If you omit *row* in the first field of a group, the macro processor uses the first line of the window; if you omit *row* in a later field specification, the macro processor continues on the line from the previous field.

The macro processor treats the first usable line of the window as row 1 (that is, it excludes the border, command line or menu bar, and message line).

Specify either *row* or *column* first.

column

specifies the column in which the macro variable or constant text begins. Each column specification consists of a pointer control and, usually, a macro expression that generates a number. These column pointer controls are available:

@*macro-expression*

specifies the column within the window given by the value of the macro expression. The macro expression must either be a positive integer or generate a positive integer.

+macro-expression

moves the pointer the number of columns given by the value of the macro expression. The macro expression must either be a positive integer or generate a positive integer.

The macro processor evaluates the macro expression when it defines the window, not when it displays the window. Thus, the column position of a field is fixed when the field is being displayed.

The macro processor treats the column after the left border as column 1. If you omit *column*, the macro processor uses column 1.

Specify either *column* or *row* first.

macro-variable

names a macro variable to be displayed or to receive the value you enter at that position. The macro variable must either be a macro variable name (not a macro variable reference) or it must be a macro expression that generates a macro variable name.

By default, you can enter or change a macro variable value when the window containing the value is displayed. To display the value without allowing changes, use the PROTECT= option.

CAUTION:

Do not overlap fields. Do not allow a field to overlap another field displayed at the same time. Unexpected results, including the incorrect assignment of values to macro variables, may occur. (Some display devices treat adjacent fields with no intervening blanks as overlapping fields.) SAS writes a warning in the SAS log if fields overlap. △

field-length

is an integer specifying how many positions in the current row are available for displaying the macro variable's value or for accepting input. The maximum value of *field-length* is the number of positions remaining in the row. You cannot extend a field beyond one row.

Note: The field length does not affect the length stored for the macro variable. The field length affects only the number of characters displayed or accepted for input in a particular field. △

If you omit *field-length* when the field contains an existing macro variable, the macro processor uses a field equal to the current length of the macro variable value, up to the number of positions remaining in the row or remaining until the next field begins.

CAUTION:

Specify a field length whenever a field contains a macro variable. If the current value of the macro variable is null, as in a macro variable defined in a %GLOBAL or %LOCAL statement, the macro processor uses a field length of 0; you cannot input any characters into the field. △

If you omit *field-length* when the macro variable is created in that field, the macro processor uses a field length of zero. Specify a field length whenever a field contains a macro variable.

'text' | "text"

contains constant text to be displayed. The text must be enclosed in either single or double quotation marks. You cannot enter a value into a field containing constant text.

options

can include the following:

ATTR=*attribute* | (*attribute-1* <. . . , *attribute-n*>) A=*attribute* |
(*attribute-1* <. . . , *attribute-n*>)

controls several display attributes of the field. The display attributes and combinations of display attributes available depend on the type of display device you use.

BLINK	causes the field to blink.
HIGHLIGHT	displays the field at high intensity.
REV_VIDEO	displays the field in reverse video.
UNDERLINE	underlines the field.

AUTOSKIP=YES | NO
AUTO=YES | NO

controls whether the cursor moves to the next unprotected field of the current window or group when you have entered data in all positions of a field. If you specify AUTOSKIP=YES, the cursor moves automatically to the next unprotected field; if you specify AUTOSKIP=NO, the cursor does not move automatically.

COLOR=*color* C=*color*

specifies a color for the field. The default color is device-dependent. *Color* can be one of these:

BLACK

BLUE

BROWN

CYAN

GRAY (or GREY)

GREEN

MAGENTA

ORANGE

PINK

WHITE

YELLOW

DISPLAY=YES | NO

determines whether the macro processor displays the characters you are entering into a macro variable value as you enter them. If you specify DISPLAY=YES (the default value), the macro processor displays the characters as you enter them. If you specify DISPLAY=NO, the macro processor does not display the characters as you enter them.

DISPLAY=NO is useful for applications that require users to enter confidental information, such as passwords. Use the DISPLAY= option only with fields containing macro variables; constant text is displayed automatically.

PROTECT=YES | NO
P=YES | NO

controls whether information can be entered into a field containing a macro variable. If you specify PROTECT=NO (the default value), you can enter information. If you specify PROTECT=YES, you cannot enter information into a field. Use the PROTECT= option only for fields containing macro variables; fields containing text are automatically protected.

REQUIRED=YES | NO
: determines whether you must enter a value for the macro variable in that field. If you specify REQUIRED=YES, you must enter a value into that field in order to remove the display. You cannot enter a null value into a required field. If you specify REQUIRED=NO (the default value), you does not have to enter a value in that field in order to remove the display. Entering a command on the command line of the window removes the effect of REQUIRED=YES.

Details Use the %WINDOW statement to define customized windows that are controlled by the macro processor. These windows have command and message lines. You can use these windows to display text and accept input. In addition, you can invoke windowing environment commands, assign function keys, and use a menu generated by the PMENU facility.

You must define a window before you can display it. The %WINDOW statement defines macro windows; the %DISPLAY statement displays macro windows. Once defined, a macro window exists until the end of the SAS session, and you can display a window or redefine it at any point.

Defining a macro window within a macro definition causes the macro processor to redefine the window each time the macro executes. If you repeatedly display a window whose definition does not change, it is more efficient to define the window outside a macro or in a macro that you execute once rather than in the macro in which you display it.

If a %WINDOW statement contains the name of a new macro variable, the macro processor creates that variable with the current scope. The %WINDOW statement creates two automatic macro variables.

SYSCMD
: contains the last command from the window's command line that was not recognized by the windowing environment.

SYSMSG
: contains text you specify to be displayed on the message line.

Note: Windowing environment file management, scrolling, searching, and editing commands are not available to macro windows. △

Example

Example 1: Creating an Application Welcome Window This %WINDOW statement creates a window with a single group of fields:

```
%window welcome color=white
           #5 @28 'Welcome to SAS.' attr=highlight
              color=blue
           #7 @15
              "You are executing Release &sysver on &sysday, &sysdate.."
           #12 @29 'Press ENTER to continue.';
```

The WELCOME window fills the entire display. The window is white, the first line of text is blue, and the other two lines are black at normal intensity. The WELCOME window does not require you to input any values. However, you must press ENTER to remove the display.

Note: Two periods are a needed delimiter for the reference to the macro variables SYSVER, SYSDAY, and SYSDATE. △

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

SAS Macro Language: Reference, Version 8

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

1-58025-522-1

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

U.S. Government Restricted Rights Notice. Use, duplication, or disclosure of the software by the government is subject to restrictions as set forth in FAR 52.227–19 Commercial Computer Software-Restricted Rights (June 1987).

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

1st printing, October 1999

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

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

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

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