CHAPTER

6

Statements

Definition

A *SAS statement* is a series of items that may include keywords, SAS names, special characters, and operators. All SAS statements end with a semicolon. A SAS statement either requests SAS to perform an operation or gives information to the system.

This book covers two kinds of SAS statements:

- those used in DATA step programming
- those that are global in scope and can be used anywhere in a SAS program.

The *SAS Procedures Guide* gives detailed descriptions of the SAS statements that are specific to each SAS procedure. *The Complete Guide to the SAS Output Delivery System* gives detailed descriptions of the Output Delivery System (ODS) statements.

DATA Step Statements

Executable and Declarative Statements

DATA step statements are those that can appear in the DATA step. They can be either executable or declarative. *Executable statements* result in some action during individual iterations of the DATA step; *declarative statements* supply information to SAS and take effect when the system compiles program statements.

The following tables show the SAS executable and declarative statements that you can use in the DATA step.

Executable Statements

ABORT	IF, Subsetting	PUT
assignment	IF-THEN/ELSE	PUT, Column
CALL	INFILE	PUT, Formatted
CONTINUE	INPUT	PUT, List
DELETE	INPUT, Column	PUT, Named
DESCRIBE	INPUT, Formatted	PUT, _ODS_
DISPLAY	INPUT, List	REDIRECT
DO	INPUT, Named	REMOVE

Executable Statements

DO, Iterative	LEAVE	REPLACE
DO Until	LINK	RETURN
DO While	LIST	SELECT
ERROR	LOSTCARD	SET
EXECUTE	MERGE	STOP
FILE	MODIFY	Sum
FILE, ODS	Null	UPDATE
GO TO	OUTPUT	

Declarative Statements

ARRAY	DATALINES	LABEL
Array Reference	DATALINES4	Labels, Statement
ATTRIB	DROP	LENGTH
BY	END	RENAME
CARDS	FORMAT	RETAIN
CARDS4	INFORMAT	WHERE
DATA	KEEP	WINDOW

DATA Step Statements by Category

In addition to being either executable or declarative, SAS DATA step statements can be grouped into four functional categories:

Table 6.1 Categories of DATA Step Statements

Statements in this category ...	let you ...
ACTION	□ create and modify variables □ select only certain observations to process in the DATA step □ look for errors in the input data □ work with observations as they are being created
CONTROL	□ skip statements for certain observations □ change the order that statements are executed □ transfer control from one part of a program to another

Statements in this category …	let you …
FILE-HANDLING	□ work with files used as input to the data set □ work with files to be written by the DATA step
INFORMATION	□ give SAS additional information about the program data vector □ give SAS additional information about the data set or data sets that are being created.

The following table lists and briefly describes the DATA step statements by category.

Table 6.2 Categories and Descriptions of DATA Step Statements

Category	Statement	Description
Action	"ABORT" on page 752	Stops executing the current DATA step, SAS job, or SAS session
	"Assignment" on page 761	Evaluates an expression and stores the result in a variable
	"CALL" on page 767	Invokes or calls a SAS CALL routine
	"DELETE" on page 783	Stops processing the current observation
	"DESCRIBE" on page 785	Retrieves source code from a stored compiled DATA step program or a DATA step view
	"ERROR" on page 800	Sets _ERROR_ to 1 and, optionally, writes a message to the SAS log
	"EXECUTE" on page 801	Executes a stored compiled DATA step program
	"IF, Subsetting" on page 847	Continues processing only those observations that meet the condition
	"LIST" on page 925	Writes to the SAS log the input data records for the observation that is being processed
	"LOSTCARD" on page 928	Resynchronizes the input data when SAS encounters a missing or invalid record in data that have multiple records per observation
	"Null" on page 949	Signals the end of data lines; acts as a placeholder
	"OUTPUT" on page 958	Writes the current observation to a SAS data set
	"REDIRECT" on page 992	Points to different input or output SAS data sets when you execute a stored program
	"REMOVE" on page 993	Deletes an observation from a SAS data set
	"REPLACE" on page 997	Replaces an observation in the same location
	"STOP" on page 1018	Stops execution of the current DATA step
	"Sum" on page 1019	Adds the result of an expression to an accumulator variable

	“WHERE” on page 1028	Selects observations from SAS data sets that meet a particular condition
Control	“CONTINUE” on page 773	Stops processing the current DO-loop iteration and resumes with the next iteration
	“DO” on page 789	Designates a group of statements to be executed as a unit
	“DO, Iterative” on page 790	Executes statements between DO and END repetitively based on the value of an index variable
	“DO UNTIL” on page 794	Executes statements in a DO loop repetitively until a condition is true
	“DO WHILE” on page 795	Executes statements repetitively while a condition is true
	“END” on page 798	Ends a DO group or a SELECT group
	“GO TO” on page 845	Moves execution immediately to the statement label that is specified
	“IF-THEN/ELSE” on page 849	Executes a SAS statement for observations that meet specific conditions
	“Labels, Statement” on page 908	Identifies a statement that is referred to by another statement
	“LEAVE” on page 909	Stops processing the current loop and resumes with the next statement in sequence
	“LINK” on page 923	Jumps to a statement label
	“RETURN” on page 1004	Stops executing statements at the current point in the DATA step and returns to a predetermined point in the step
	“SELECT” on page 1007	Executes one of several statements or groups of statements
File-handling	“BY” on page 765	Controls the operation of a SET, MERGE, MODIFY, or UPDATE statement in the DATA step and sets up special grouping variables
	“CARDS” on page 768	Indicates that data lines follow
	“CARDS4” on page 768	Indicates that data lines that contain semicolons follow
	“DATA” on page 774	Begins a DATA step and provides names for any output SAS data sets
	“DATALINES” on page 781	Indicates that data lines follow
	“DATALINES4” on page 782	Indicates that data lines that contain semicolons follow
	“FILE” on page 802	Specifies the current output file for PUT statements
	“FILE, ODS” on page 815	Defines the structure of the data component that holds the results of the DATA step and binds that component to a template to produce an output object. ODS sends this object to all open ODS destinations, each of which formats the object appropriately. Also controls what happens when the PUT statement tries to write past the end of a line.
	“INFILE” on page 857	Identifies an external file to read with an INPUT statement

	“INPUT” on page 876	Describes the arrangement of values in the input data record and assigns input values to the corresponding SAS variables
	“INPUT, Column” on page 890	Reads input values from specified columns and assigns them to the corresponding SAS variables
	“INPUT, Formatted” on page 893	Reads input values with specified informats and assigns them to the corresponding SAS variables
	“INPUT, List” on page 897	Scans the input data record for input values and assigns them to the corresponding SAS variables
	“INPUT, Named” on page 902	Reads data values that appear after a variable name that is followed by an equal sign and assigns them to corresponding SAS variables
	“MERGE” on page 930	Joins observations from two or more SAS data sets into single observations
	“MODIFY” on page 933	Replaces, deletes, and appends observations in an existing SAS data set in place; does not create an additional copy
	“PUT” on page 962	Writes lines to the SAS log, to the SAS procedure output file, or to an external file that is specified in the most recent FILE statement
	“PUT, Column” on page 977	Writes variable values in the specified columns in the output line
	“PUT, Formatted” on page 979	Writes variable values with the specified format in the output line
	“PUT, List” on page 983	Writes variable values and the specified character strings in the output line
	“PUT, Named” on page 987	Writes variable values after the variable name and an equal sign
	“PUT, _ODS_” on page 989	Writes data values to a special buffer from which they can be written to the data component, and formatted by ODS destinations
	“SET” on page 1010	Reads an observation from one or more SAS data sets
	“UPDATE” on page 1023	Updates a master file by applying transactions
Information	“ARRAY” on page 755	Defines elements of an array
	“Array Reference” on page 759	Describes the elements in an array to be processed
	“ATTRIB” on page 762	Associates a format, informat, label, and/or length with one or more variables
	“DROP” on page 796	Excludes variables from output SAS data sets
	“FORMAT” on page 843	Associates formats with variables
	“INFORMAT” on page 873	Associates informats with variables
	“KEEP” on page 905	Includes variables in output SAS data sets
	“LABEL” on page 906	Assigns descriptive labels to variables
	“LENGTH” on page 910	Specifies the number of bytes for storing variables

"MISSING" on page 932	Assigns characters in your input data to represent special missing values for numeric data
"RENAME" on page 995	Specifies new names for variables in output SAS data sets
"RETAIN" on page 999	Causes a variable that is created by an INPUT or assignment statement to retain its value from one iteration of the DATA step to the next

*

Global Statements

Definition

Global statements generally provide information to SAS, request information or data, move between different modes of execution, or set values for system options. Other global statements (ODS statements) deliver output in a variety of formats, such as in Hypertext Markup Language (HTML). You can use global statements anywhere in a SAS program. Global statements are not executable; they take effect as soon as SAS compiles program statements.

Other SAS software products have additional global statements that are used with those products. For information, see the SAS documentation for those products.

Global Statements by Category

The following table lists and describes SAS global statements, organized by function into five categories:

Table 6.3 Global Statements by Category

Statements in this category ...	let you ...
DATA ACCESS	associate reference names with SAS data libraries, SAS catalogs, external files and output devices, and access remote files.
OPERATING ENVIRONMENT	access the operating environment directly.
LOG CONTROL	alter the appearance of the SAS log.
OUTPUT CONTROL	add titles and footnotes to your SAS output; deliver output in a variety of formats.
PROGRAM CONTROL	govern the way SAS processes your SAS program.
WINDOW DISPLAY	display and customize windows.

The following table provides brief descriptions of SAS global statements. For more detailed information, see the individual statements.

* The null statement is considered an executable statement in the DATA step because you can branch to it. However, the null statement can appear anywhere in a SAS program.

Table 6.4 Categories and Descriptions of DATA Step Statements

Category	Statement	Description
Data Access	"CATNAME" on page 769	Logically combines two or more catalogs into one by associating them with a catref (a shortcut name); clears one or all catrefs; lists the concatenated catalogs in one concatenation or in all concatenations
	"FILENAME" on page 821	Associates a SAS fileref with an external file or an output device; disassociates a fileref and external file; lists attributes of external files
	"FILENAME, CATALOG Access Method" on page 826	References a SAS catalog as an external file
	"FILENAME, FTP Access Method" on page 829	Allows you to access remote files using the FTP protocol
	"FILENAME, SOCKET Access Method" on page 835	Allows you to read from or write to a TCP/IP socket
	"FILENAME, URL Access Method" on page 839	Allows you to access remote files using the URL access method
	"LIBNAME" on page 913	Associates or disassociates a SAS data library with a libref (a shortcut name); clears one or all librefs; lists the characteristics of a SAS data library; concatenates SAS data libraries; implicitly concatenates SAS catalogs.
	"LIBNAME, SAS/ ACCESS" on page 919	Associates a SAS libref with a database management system (DBMS) database, schema, server, or group of tables or views
Log Control	"Comment" on page 772	Documents the purpose of the statement or program
	"PAGE" on page 961	Skips to a new page in the SAS log
	"SKIP" on page 1017	Creates a blank line in the SAS log
Operating Environment	"X" on page 1043	Issues an operating-environment command from within a SAS session
Output Control	"FOOTNOTE" on page 841	Prints up to ten lines of text at the bottom of the procedure or DATA step output
	"ODS EXCLUDE" on page 950	Specifies output objects to exclude from ODS destinations
	"ODS HTML" on page 951	Opens, manages, or closes the HTML destination. If the destination is open, you can create HTML output (output that is written in Hypertext Markup Language).
	"ODS LISTING" on page 952	Opens, manages, or closes the Listing destination
	"ODS OUTPUT" on page 953	Creates a SAS data set from an output object and manages the selection and exclusion lists for the Output destination

	“ODS PATH” on page 954	Specifies which locations to search for definitions that were created by PROC EMPLATE, as well as the order in which to search for them
	“ODS PRINTER” on page 954	Opens, manages, or closes the Printer destination. If the destination is open, you can create Printer output (output that is formatted for a high-resolution printer)
	“ODS SELECT” on page 955	Specifies output objects for ODS destinations
	“ODS SHOW” on page 956	Writes to the SAS log the specified selection or exclusion list
	“ODS TRACE” on page 956	Writes to the SAS log a record of each output object that is created, or suppresses the writing of this record
	“ODS VERIFY” on page 957	Prints or suppresses a warning that a style definition or a table definition that is used is not supplied by SAS Institute
	“TITLE” on page 1020	Specifies title lines for SAS output
Program Control	“DM” on page 787	Submits SAS Program Editor, Log, Procedure Output or text editor commands as SAS statements
	“ENDSAS” on page 800	Terminates a SAS job or session after the current DATA or PROC step executes
	“%INCLUDE” on page 851	Brings a SAS programming statement, data lines, or both, into a current SAS program
	“%LIST” on page 926	Displays lines that are entered in the current session
	“OPTIONS” on page 957	Changes the value of one or more SAS system options
	“RUN” on page 1005	Executes the previously entered SAS statements
	“%RUN” on page 1006	Ends source statements following a %INCLUDE * statement
Window Display	“DISPLAY” on page 785	Displays a window that is created with the WINDOW statement
	“WINDOW” on page 1033	Creates customized windows for your applications

Dictionary

ABORT

Stops executing the current DATA step, SAS job, or SAS session

Valid: in a DATA step

Category: Action

Type: Executable

Syntax

ABORT <ABEND | RETURN> <*n*>;

Without Arguments

If you specify no argument, the ABORT statement produces these results under the following methods of operation:

batch mode and noninteractive mode
- □ stops processing the current DATA step and writes an error message to the SAS log. Data sets can contain an incomplete number of observations or no observations, depending on when SAS encountered the ABORT statement.
- □ sets the OBS= system option to 0.
- □ continues limited processing of the remainder of the SAS job, including executing macro statements, executing system options statements, and syntax checking of program statements.
- □ creates output data sets for subsequent DATA and PROC steps with no observations.

windowing environment
- □ stops processing the current DATA step
- □ creates a data set that contains the observations that are processed before the ABORT statement is encountered
- □ prints a message to the log that an ABORT statement terminated the DATA step
- □ continues processing any DATA or PROC steps that follow the ABORT statement.

interactive line mode
stops processing the current DATA step. Any further DATA steps or procedures execute normally.

Arguments

ABEND
causes abnormal termination of the current SAS job or session. Results depend on the method of operation:
- □ batch mode and noninteractive mode
  - □ stops processing immediately
  - □ sends an error message to the SAS log that states that execution was terminated by the ABEND option of the ABORT statement
  - □ does not execute any subsequent statements or check syntax
  - □ returns control to the operating environment; further action is based on how your operating environment and your site treat jobs that end abnormally.
- □ windowing environment and interactive line mode
  - □ causes your windowing environment and interactive line mode to stop processing immediately and return you to your operating environment.

RETURN
causes the immediate normal termination of the current SAS job or session. Results depend on the method of operation:

- batch mode and noninteractive mode
  - stops processing immediately
  - sends an error message to the SAS log stating that execution was terminated by the RETURN option of the ABORT statement
  - does not execute any subsequent statements or check syntax
  - returns control to your operating environment with a condition code indicating an error
- windowing environment and interactive line mode
  - causes your windowing environment and interactive line mode to stop processing immediately and return you to your operating environment.

n
is an integer value that enables you to specify a condition code that SAS returns to the operating environment when it stops executing.

Operating Environment Information: The range of values for *n* depends on your operating environment. △

Details

The ABORT statement causes SAS to stop processing the current DATA step. What happens next depends on

- the method you use to submit your SAS statements
- the arguments you use with ABORT
- your operating environment.

The ABORT statement usually appears in a clause of an IF-THEN statement or a SELECT statement that is designed to stop processing when an error condition occurs.

Note: When you execute an ABORT statement in a DATA step, SAS does not use data sets that were created in the step to replace existing data sets with the same name. △

Operating Environment Information: The only difference between the ABEND and RETURN options is that with ABEND further action is based on how your operating environment and site treat jobs that end abnormally. RETURN simply returns a condition code that indicates an error. △

Comparisons

- When you use the SAS windowing environment or interactive line mode, the ABORT statement and the STOP statement both stop processing. The ABORT statement sets the value of the automatic variable _ERROR_ to 1, and the STOP statement does not.
- In batch or noninteractive mode, the ABORT and STOP statements also have different effects. Both stop processing, but only ABORT sets the value of the automatic variable _ERROR_ to 1. Use the STOP statement, therefore, when you want to stop only the current DATA step and continue processing with the next step.

Examples

This example uses the ABORT statement as part of an IF-THEN statement to stop execution of SAS when it encounters a data value that would otherwise cause a division-by-zero condition.

```
if volume=0 then abort 255;
   density=mass/volume;
```

The *n* value causes SAS to return the condition code 255 to the operating environment when the ABORT statement executes.

See Also

Statement:

"STOP" on page 1018

ARRAY

Defines elements of an array

Valid: in a DATA step
Category: Information
Type: Declarative

Syntax

ARRAY *array-name* { *subscript* } <$><*length*>
<*array-elements*> <(*initial-value-list*)>;

Arguments

array-name
names the array.

Restriction: *Array-name* must be a SAS name that is not the name of a SAS variable in the same DATA step.

CAUTION:
Using the name of a SAS function as an array name can cause unpredictable results. If you inadvertently use a function name as the name of the array, SAS treats parenthetical references that involve the name as array references, not function references, for the duration of the DATA step. A warning message is written to the SAS log. △

{subscript}
describes the number and arrangement of elements in the array by using an asterisk, a number, or a range of numbers. *Subscript* has one of these forms:

{*dimension-size(s)*}
indicates the number of elements in each dimension of the array. *Dimension-size* is a numeric representation of either the number of elements in a one-dimensional array or the number of elements in each dimension of a multidimensional array.

Tip: In this book, the subscript is enclosed in braces ({}). Brackets ([ ])and parentheses (( ))are also allowed.

Example: An array with one dimension can be defined as

```
array simple{3} red green yellow;
```

This ARRAY statement defines an array that is named SIMPLE that groups together three variables that are named RED, GREEN, and YELLOW.

Example: An array with more than one dimension is known as a multidimensional array. You can have any number of dimensions in a multidimensional array. For example, a two-dimensional array provides row and column arrangement of array elements. This statement defines a two-dimensional array with five rows and three columns:

```
array x{5,3} score1-score15;
```

SAS places variables into a two-dimensional array by filling all rows in order, beginning at the upper-left corner of the array (known as row-major order).

{<*lower*:>*upper*<, . . .<*lower*:> *upper*>}
are the bounds of each dimension of an array, where *lower* is the lower bound of that dimension and *upper* is the upper bound.

Range: In most explicit arrays, the subscript in each dimension of the array ranges from 1 to *n*, where *n* is the number of elements in that dimension.

Example: In the following example, the value of each dimension is by default the upper bound of that dimension.

```
array x{5,3} score1-score15;
```

As an alternative, the following ARRAY statement is a longhand version of the previous example:

```
array x{1:5,1:3} score1-score15;
```

Tip: For most arrays, 1 is a convenient lower bound; thus, you do not need to specify the lower and upper bounds. However, specifying both bounds is useful when the array dimensions have a convenient beginning point other than 1.

Tip: To reduce the computational time that is needed for subscript evaluation, specify a lower bound of 0.

{*}
indicates that SAS is to determine the subscript by counting the variables in the array. When you specify the asterisk, also include *array-elements*.

Restriction: You cannot use the asterisk with _TEMPORARY_ arrays or when you define a multidimensional array.

$
indicates that the elements in the array are character element.

Tip: The dollar sign is not necessary if the elements have been previously defined as character elements.

length
specifies the length of elements in the array that have not been previously assigned a length.

array-elements
names the elements that make up the array. *Array-elements* must be either all numeric or all character, and they can be listed in any order. The elements can be

variables
lists variable names.

Range: The names must be either variables that you define in the ARRAY statement or variables that SAS creates by concatenating the array name and a number. For instance, when the subscript is a number (not the asterisk), you do not need to name each variable in the array. Instead, SAS creates variable names by concatenating the array name and the numbers , 2, 3, . . . *n*.

Tip: These SAS variable lists enable you to reference variables that have been previously defined in the same DATA step:

NUMERIC
indicates all numeric variables.

CHARACTER
indicates all character variables.

ALL
indicates all variables.

Restriction: If you use _ALL_, all the previously defined variables must be of the same type.

Featured in: Example 1 on page 758

TEMPORARY
creates a list of temporary data elements.

Range: Temporary data elements can be numeric or character.

Tip: Temporary data elements behave like DATA step variables with these exceptions:

- They do not have names. Refer to temporary data elements by the array name and dimension.
- They do not appear in the output data set.
- You cannot use the special subscript asterisk (*) to refer to all the elements.
- Temporary data element values are always automatically retained, rather than being reset to missing at the beginning of the next iteration of the DATA step.

Tip: Arrays of temporary elements are useful when the only purpose for creating an array is to perform a calculation. To preserve the result of the calculation, assign it to a variable. You can improve performance time by using temporary data elements.

(*initial-value-list*)
gives initial values for the corresponding elements in the array. The values for elements can be numbers or character strings. You must enclose all character strings in quotation marks. To specify one or more initial values directly, use the following format:

(*initial-value(s)*)

To specify an iteration factor and nested sublists for the initial values, use the following format:

<constant-iter-value> <(>*constant value* | *constant-sublist*<)>

Restriction: If you specify both an *initial-value-list* and *array-elements*, then *array-elements* must be listed before *initial-value-list* in the ARRAY statement.

Tip: You can assign initial values to both variables and temporary data elements.

Tip: Elements and values are matched by position. If there are more array elements than initial values, the remaining array elements receive missing values and SAS issues a warning. See Example 2 on page 758, and Example 3 on page 759.

Tip: You can separate the values in the initial value list with either a comma or a blank space.

Tip: If you have not previously specified the attributes of the array elements (such as length or type), the attributes of any initial values that you specify are automatically assigned to the corresponding array element.

Note: Initial values are retained until a new value is assigned to the array element. △

Tip: When any (or all) elements are assigned initial values, all elements behave as if they had been named one a RETAIN statement.

Examples: The following examples show how to use the iteration factor and nested sublists. All of these ARRAY statements contain the same initial value list:

- `ARRAY x{10} x1-x10 (10*5);`
- `ARRAY x{10} x1-x10 (5*(5 5));`
- `ARRAY x{10} x1-x10 (5 5 3*(5 5) 5 5);`
- `ARRAY x{10} x1-x10 (2*(5 5) 5 5 2*(5 5));`
- `ARRAY x{10} x1-x10 (2*(5 2*(5 5)));`

Details

The ARRAY statement defines a set of elements that you plan to process as a group. You refer to elements of the array by the array name and subscript. Because you usually want to process more than one element in an array, arrays are often referenced within DO groups.

Comparisons

- Arrays in the SAS language are different from those in many other languages. A SAS array is simply a convenient way of temporarily identifying a group of variables. It is not a data structure, and *array-name* is not a variable.
- An ARRAY statement defines an array. An array reference uses an array element in a program statement.

Examples

Example 1: Defining Arrays

- `array rain {5} janr febr marr aprr mayr;`
- `array days{7} d1-d7;`
- `array month{*} jan feb jul oct nov;`
- `array x{*} _NUMERIC_;`
- `array qbx{10};`
- `array meal{3};`

Example 2: Assigning Initial Numeric Values

- `array test{4} t1 t2 t3 t4 (90 80 70 70);`
- `array test{4} t1-t4 (90 80 2*70);`
- `array test{4} _TEMPORARY_ (90 80 70 70);`

Example 3: Defining Initial Character Values

□ `array test2{*} a1 a2 a3 ('a','b','c');`

Example 4: Defining More Advanced Arrays

□ `array new{2:5} green jacobs denato fetzer;`

□ `array x{5,3} score1-score15;`

□ `array test{3:4,3:7} test1-test10;`

□ `array temp{0:999} _TEMPORARY_;`

See Also

Statement:

"Array Reference" on page 759

Array Processing in *SAS Language Reference: Concepts*

Array Reference

Describes the elements in an array to be processed

Valid: in a DATA step
Category: Information
Type: Declarative

Syntax

array-name { *subscript* }

Arguments

array-name
is the name of an array that was previously defined with an ARRAY statement in the same DATA step.

{subscript}
specifies the subscript. Any of these forms can be used:

{*variable-1*< , . . . *variable-n*>}
indicates a variable, or variable list that is usually used with DO-loop processing. For each execution of the DO loop, the current value of this variable becomes the subscript of the array element being processed. See Example 1 on page 761 .

Tip: In this book, the subscript is enclosed in braces ({ }). You can also use brackets ([ ])or parentheses (( )).

{*}
forces SAS to treat the elements in the array as a variable list.

Tip: The asterisk can be used with the INPUT and PUT statements, and with some SAS functions.

Tip: This syntax is provided for convenience and is an exception to usual array processing.

Restriction: When you define an array that contains temporary array elements, you cannot reference the array elements with an asterisk. See Example 4 on page 761 .

expression-1< , . . . *expression-n*>
indicates a SAS expression.

Range: The expression must evaluate to a subscript value when the statement that contains the array reference executes. The expression can also be an integer with a value between the lower and upper bounds of the array, inclusive. See Example 3 on page 761 .

Details

- To refer to an array in a program statement, use an array reference. The ARRAY statement that defines the array must appear in the DATA step before any references to that array. An array definition is only in effect for the duration of the DATA step. If you want to use the same array in several DATA steps, redefine the array in each step.

CAUTION:
Using the name of a SAS function as an array name can cause unpredictable results. If you inadvertently use a function name as the name of the array, SAS treats parenthetical references that involve the name as array references, not function references, for the duration of the DATA step. A warning message is written to the SAS log. △

- You can use an array reference anywhere that you can write a SAS expression, including SAS functions and these SAS statements:
  - assignment statement
  - sum statement
  - DO UNTIL(*expression*)
  - DO WHILE(*expression*)
  - IF
  - INPUT
  - PUT
  - SELECT
  - WINDOW.
- The DIM function is often used with the iterative DO statement to return the number of elements in a dimension of an array, when the lower bound of the dimension is 1. If you use DIM, you can change the number of array elements without changing the upper bound of the DO statement. For example, because DIM(NEW) returns a value of 4, the following statements process all the elements in the array:

```
array new{*} score1-score4;
   do i=1 to dim(new);
      new{i}=new{i}+10;
   end;
```

Comparisons

- An ARRAY statement defines an array, whereas an array reference processes members of the array.

Examples

Example 1: Using Iterative DO-Loop Processing In this example, the statements process each element of the array, using the value of variable I as the subscript on the array references for each iteration of the DO loop. If an array element has a value of 99, the IF-THEN statement changes that value to 100.

```
array days{7} d1-d7;
   do i=1 to 7;
      if days{i}=99 then days{i}=100;
   end;
```

Example 2: Referencing Many Arrays in One Statement You can refer to more than one array in a single SAS statement. In this example, you create two arrays, DAYS and HOURS. The statements inside the DO loop substitute the current value of variable I to reference each array element in both arrays.

```
array days{7} d1-d7;
   array hours{7} h1-h7;
      do i=1 to 7;
         if days{i}=99 then days{i}=100;
         hours{i}=days{i}*24;
      end;
```

Example 3: Specifying the Subscript In this example, the INPUT statement reads in variables A1, A2, and the third element (A3) of the array named ARR1:

```
array arr1{*} a1-a3;
x=1;
input a1 a2 arr1{x+2};
```

Example 4: Using the Asterisk References as a Variable List

□
```
array cost{10} cost1-cost10;
totcost=sum(of cost {*});
```

□
```
array days{7} d1-d7;
input days {*};
```

□
```
array hours{7} h1-h7;
put hours {*};
```

See Also

Function:

"DIM" on page 330

Statements

"ARRAY" on page 755

"DO, Iterative" on page 790

Array Processing in *SAS Language Reference: Concepts*

Assignment

Evaluates an expression and stores the result in a variable

Valid: in a DATA step
Category: Action
Type: Executable

Syntax

variable=expression;

Arguments

variable
names a new or existing variable.

Range: *Variable* can be a variable name, array reference, or SUBSTR function.

Tip: Variables that are created by the Assignment statement are not automatically retained.

expression
is any SAS expression.

Tip: *expression* can contain the variable that is used on the left side of the equal sign. When a variable appears on both sides of a statement, the original value on the right side is used to evaluate the expression, and the result is stored in the variable on the left side of the equal sign. For more information, see "Expressions" in *SAS Language Reference: Concepts*.

Details

Assignment statements evaluate the expression on the right side of the equal sign and store the result in the variable that is specified on the left side of the equal sign.

Examples

These assignment statements use different kinds of expressions:

- `name='Amanda Jones';`
- `WholeName='Ms. '||name;`
- `a=a+b;`

See Also

Statement:

"Sum" on page 1019

ATTRIB

Associates a format, informat, label, and/or length with one or more variables

Valid: in a DATA step

Category: Information
Type: Declarative

Syntax

ATTRIB *variable-list(s) attribute-list(s)* ;

Arguments

variable-list
names the variables that you want to associate with the attributes.

Tip: List the variables in any form that SAS allows.

attribute-list
specifies one or more attributes to assign to *variable-list*. Specify one or more of these attributes in the ATTRIB statement:

FORMAT=*format*
associates a format with variables in *variable-list*.

Tip: The format can be either a standard SAS format or a format that is defined with the FORMAT procedure.

INFORMAT=*informat*
associates an informat with variables in *variable-list*.

Tip: The informat can be either a standard SAS informat or an informat that is defined with the FORMAT procedure.

LABEL='*label*'
associates a label with variables in *variable-list*.

LENGTH=<$>*length*
specifies the length of variables in *variable-list*.

Requirement: Put a dollar sign ($) in front of the length of character variables.

Tip: Use the ATTRIB statement before the SET statement to change the length of variables in an output data set when you use an existing data set as input.

Range: For character variables, the range is 1 to 32,767 for all operating environments.

Operating Environment Information: For numeric variables, the minimum length you can specify with the LENGTH= specification is 2 in some operating environments and 3 in others. △

Details

The Basics Using the ATTRIB statement in the DATA step permanently associates attributes with variables by changing the descriptor information of the SAS data set that contains the variables.

You can use ATTRIB in a PROC step, but the rules are different.

How SAS Treats Variables when You Assign Informats with the INFORMAT= Option on the ATTRIB Statement Informats that are associated with variables by using the INFORMAT= option on the ATTRIB statement behave like informats that are used

with modified list input. SAS reads the variables by using the scanning feature of list input, but applies the informat. In modified list input, SAS

- does not use the value of *w* in an informat to specify column positions or input field widths in an external file
- uses the value of *w* in an informat to specify the length of previously undefined character variables
- ignores the value of *w* in numeric informats
- uses the value of *d* in an informat in the same way it usually does for numeric informats
- treats blanks that are embedded as input data as delimiters unless you change their status with a DELIMITER= option specification in an INFILE statement.

If you have coded the INPUT statement to use another style of input, such as formatted input or column input, that style of input is not used when you use the INFORMAT= option on the ATTRIB statement.

Comparisons

You can use either an ATTRIB statement or an individual attribute statement such as FORMAT, INFORMAT, LABEL, and LENGTH to change an attribute that is associated with a variable.

Examples

Here are examples of ATTRIB statements that contain

- single variable and single attribute:

```
attrib cost length=4;
```

- single variable with multiple attributes:

```
attrib saleday informat=mmddyy.
format=worddate.;
```

- multiple variables with the same multiple attributes:

```
attrib x y length=$4 label='TEST VARIABLE';
```

- multiple variables with different multiple attributes:

```
attrib x length=$4 label='TEST VARIABLE'
       y length=$2 label='RESPONSE';
```

- variable list with single attribute:

```
attrib month1-month12
       label='MONTHLY SALES';
```

See Also

Statements:

"FORMAT" on page 843
"INFORMAT" on page 873
"LABEL" on page 906
"LENGTH" on page 910

BY

Controls the operation of a SET, MERGE, MODIFY, or UPDATE statement in the DATA step and sets up special grouping variables

Valid: in a DATA step or a PROC step
Category: File-handling
Type: Declarative

Syntax

BY <DESCENDING> <GROUPFORMAT> *variable-1*
<. . .<DESCENDING> <GROUPFORMAT>
variable-n> <NOTSORTED>;

Arguments

DESCENDING
indicates that the data sets are sorted in descending order by the variable that is specified. DESCENDING means largest to smallest numerically, or reverse alphabetical for character variables.

Restriction: You cannot use the DESCENDING option with indexed data sets because indexes are always stored in ascending order.

Featured in: Example 2 on page 767

GROUPFORMAT
uses the formatted values, instead of the stored values, of the variable when you reference FIRST.*variable* and LAST.*variable* in a DATA step.

Restriction: You must sort the observations in a data set based on the value of the BY variables before using GROUPFORMAT in the BY statement.

Restriction: You cannot use the GROUPFORMAT option in a BY statement in a PROC step.

Tip: GROUPFORMAT is useful when you define your own formats to display grouped data.

Tip: Using GROUPFORMAT in the DATA step ensures that BY groups that you use to create a data set match those in PROC steps that report grouped, formatted data.

Interaction: If you also use the NOTSORTED option, you can group the observations in a data set by the formatted value of the BY variables instead of by sorting them.

Comparison: BY-group processing in the DATA step using GROUPFORMAT is the same as BY-group processing with formatted values in SAS procedures.

See Also: Data grouped by formatted values in "BY-Group Processing" in *SAS Language Reference: Concepts*.

Featured in: Example 4 on page 767

variable

names each variable by which the data set is sorted or indexed. These variables are referred to as BY variables for the current DATA or PROC step.

Tip: The data set can be sorted or indexed by more than one variable.

Featured in: Example 1 on page 766, Example 2 on page 767, Example 3 on page 767, and Example 4 on page 767

NOTSORTED

specifies that observations with the same BY value are grouped together but are not necessarily sorted in alphabetical or numeric order.

Restriction: You cannot use NOTSORTED with the MERGE and UPDATE statements.

Tip: The NOTSORTED option can appear anywhere in the BY statement.

Tip: NOTSORTED is useful if you have data that fall into other logical groupings such as chronological order or categories.

Featured in: Example 3 on page 767

Details

In a DATA Step The BY statement applies only to the SET, MERGE, MODIFY, or UPDATE statement that precedes it in the DATA step, and only one BY statement can accompany each of these statements in a DATA step.

The data sets that are listed in the SET, MERGE, or UPDATE statements must be sorted by the values of the variables that are listed in the BY statement or have an appropriate index. As a default, SAS expects the data sets to be arranged in ascending numeric order or in alphabetical order. The observations can be arranged by

- sorting the data set
- creating an index for the variables
- inputting the observations in order.

Note: MODIFY does not require sorted data, but sorting can improve performance. △

In a PROC Step You can specify the BY statement with some SAS procedures to modify their action. Refer to the individual procedure in *SAS Procedures Guide* for a discussion of how the BY statement affects processing for SAS procedures.

BY-Group Processing For a complete explanation of how SAS processes grouped data and of how to prepare your data, see “By-Group Processing” in *SAS Language Reference: Concepts*.

With SAS Data Views If you are using SAS data views, refer to the appropriate SAS documentation for your database management system before you use the BY statement.

Examples

Example 1: Specifying One or More BY Variables

- Observations are in ascending order of the variable DEPT:

```
by dept;
```

- Observations are in alphabetical (ascending) order by CITY and, within each value of CITY, in ascending order by ZIPCODE:

```
by city zipcode;
```

Example 2: Specifying Sort Order

- Observations are in ascending order of SALESREP and, within each SALESREP value, in descen
- Observations are in descending order of BEDROOMS, and, within each value of BEDROOMS, in descending order of PRICE:

```
by descending bedrooms descending price;
```

Example 3: BY-Processing with Nonsorted Data Observations are ordered by the name of the month in which the expenses were accrued:

```
by month notsorted;
```

Example 4: Grouping Observations By Using Formatted Values Use PROC FORMAT to create the user-defined format RANGE:

```
proc format;
   value range 1-2='LOW' 3-4='MEDIUM' 5-6='HIGH';
run;
```

Create a data set ordered by formatted values. TEST must already be ordered by the *stored* values of SCORE. BY GROUPFORMAT causes TEST to be processed in BY groups based on the *formatted* values of SCORE and creates NEWTEST in this order:

```
data newtest;
   set test;
   format score range.;
   by groupformat score;
run;
```

PROC PRINT uses the format RANGE to write the values of SCORE:

```
proc print data=newtest;
   var name score
   by score;
   format score range.;
run;
```

See Also

Statements:

"MERGE" on page 930
"MODIFY" on page 933
"SET" on page 1010
"UPDATE" on page 1023

CALL

Invokes or calls a SAS CALL routine

Valid: in a DATA step
Category: Action
Type: Executable

Syntax

CALL *routine*(*parameter-1*<, . . . *parameter-n*>);

Arguments

routine
names the SAS CALL routine that you want to invoke. For information on available routines, see Chapter 4, "Functions and CALL Routines," on page 199.

(*parameter*)
is a piece of information to be passed to or returned from the routine.

Requirement: Enclose this information, which depends on the specific routine, in parentheses.

Tip: You can specify additional parameters, separated by commas.

Details

SAS CALL routines can assign variable values and perform other system functions.

See

Chapter 4, "Functions and CALL Routines," on page 199

CARDS

Indicates that data lines follow

Valid: in a DATA step
Category: File-handling
Type: Declarative
Aliases: DATALINES, LINES
See: "DATALINES" on page 781

CARDS4

Indicates that data lines that contain semicolons follow

Valid: in a DATA step
Category: File-handling
Type: Declarative
Aliases: DATALINES4, LINES4
See: "DATALINES4" on page 782

CATNAME

Logically combines two or more catalogs into one by associating them with a catref (a shortcut name); clears one or all catrefs; lists the concatenated catalogs in one concatenation or in all concatenations

Valid: Anywhere
Category: Data Access
See Also: LIBNAME statement; FILENAME, CATALOG statement

Syntax

❶**CATNAME**<*libref.*> *catref*
< *(libref-1.catalog-1* <(ACCESS=READONLY)>
<...*libref-n.catalog–n* <(ACCESS=READONLY)>)> ;
❷**CATNAME** <*libref.*> *catref* CLEAR | _ALL_ CLEAR;
❸**CATNAME** <*libref.*> *catref* LIST | _ALL_ LIST;

Arguments

libref
is any previously-assigned SAS libref. If you do not specify a libref, SAS concatenates the catalog in the work library, using the catref that you specify.

Restriction: The libref must have been previously assigned.

catref
is a unique catalog reference name for a catalog or a catalog concatenation that is specified in the statement. Separate the catref from the libref with a period, as in *libref.catref*. Any SAS name may be used for this catref.

catalog
is the name of a catalog to be made available for use in the catalog concatenation.

Options

CLEAR
disassociates a currently assigned *catref* or *libref.catref*.

Tip: Specify a specific *catref* or *libref.catref* to disassociate it from a single concatenation. Specify _ALL_ CLEAR to disassociate all currently assigned *catref* or *libref.catref* concatenations.

ALL CLEAR
disassociates all currently assigned *catref* or *libref.catref* concatenations.

LIST
writes the catalog names that are included in the specified concatenation to the SAS log.

Tip: Specify *catref* or *libref.catref* to list the attributes of a single concatenation. Specify _ALL_ to list the attributes of all catalog concatenations in your current session.

ALL LIST
writes all catalog names that are included in any current catalog concatenation to the SAS log.

ACCESS=READONLY
assigns a read-only attribute to the catalog. SAS, therefore, will allow users to read from the catalog entries but not to update information or to write new information.

Details

Why Use CATNAME? CATNAME is useful because it allows you to access entries in multiple catalogs by specifying a single catalog reference name (*libref.catref* or just *catref*). After you create a catalog concatenation, you can specify the catref in any context that accepts a simple (nonconcatenated) catref.

Rules for Catalog Concatenation To use catalog concatenation effectively, you must understand the rules that determine how catalog entries are located among the concatenated catalogs:

1. When a catalog entry is opened for input or update, the concatenated catalogs are searched and the first occurrence of the specified entry is used.
2. When a catalog entry is opened for output, it will be created in the first catalog that is listed in the concatenation.

   Note: A new catalog entry is created in the first catalog even if there is an entry with the same name in another part of the concatenation. △

   Note: If the first catalog in a concatenation that is opened for update does not exist, the item will be written to the next catalog that exists in the concatenation. △
3. When you want to delete or rename a catalog entry, only the first occurrence of the entry is affected.
4. Any time a list of catalog entries is displayed, only one occurrence of a catalog entry name is shown.

   Note: Even if the name occurs multiple times in the concatenation, only the first occurrence is shown. △

Comparisons

- The CATNAME statement is like a LIBNAME statement for catalogs. The LIBNAME statement allows you to assign a shortcut name to a SAS data library so that you can use the shortcut name to find the files and use the data they

contain. CATNAME allows you to assign a short name <*libref.*>*catref* (libref is optional) to one or more catalogs so that SAS can find the catalogs and use all or some of the entries in each catalog.

- The CATNAME statement *explicitly* concatenates SAS catalogs. You can use the LIBNAME statement to *implicitly* concatenate SAS catalogs.

Examples

Example 1: Assigning and Using a Catalog Concatenation As an example, you may need to access entries in several SAS catalogs. The most efficient way to access the information is to logically concatenate the catalogs. This allows access to the information without actually creating a new, separate, possibly very large catalog.

Assign libnames to the SAS data libraries that contain the catalogs that you want to concatenate:

```
libname mylib1 'my-data-library-1';
libname mylib2 'my-data-library-2';
```

Assign a catref, which can be any valid SAS name, to the list of catalogs that you want to logically concatenate:

```
catname allcats (mylib1.catalog1 mylib2.catalog2);
```

Because no libref is specified, the libref is WORK by default. When you want to access a catalog entry in either of these catalogs, use the libref WORK and the catalog reference name ALLCATS instead of the original librefs and catalog names. For example, to access a catalog entry named APPKEYS.KEYS in the catalog MYLIB1.CATALOG1, specify

```
work.allcats.appkeys.keys
```

Example 2: Creating a Nested Catalog Concatenation Once you have created a concatenated catalog, you can use CATNAME to combine your concatenation with other single catalogs or other concatenated catalogs. This is useful, because you can use a single catref to access many different catalog combinations.

```
libname local 'my_dir';
libname main  'public_dir';


catname private_catalog (local.my_application_code
                         local.my_frames
                         local.my_formats);

catname combined_catalogs (private_catalog
                           main.public_catalog);
```

In the above example, an application developer could be working on private copies of his/her application entries by using PRIVATE_CATALOG. If the same user wanted to see how his/her entries functioned when they were combined with the public version of the application, that user could find out by using COMBINED_CATALOGS.

See Also

Statements:

Chapter 6, “Statements,” on page 743
“FILENAME, CATALOG Access Method” on page 826
“LIBNAME” on page 913 for a discussion of *implicitly* concatenating SAS catalogs.

Comment

Documents the purpose of the statement or program

Valid: anywhere
Category: Log Control

Syntax

**message*;
or
/**message**/

Arguments

****message*;**
specifies the text that explains or documents the statement or program.

Range: These comments can be any length.

Restriction: These comments must be written as separate statements.

Restriction: These comments cannot contain internal semicolons.

/*message*/
specifies the text that explains or documents the statement or program.

Range: These comments can be any length.

Restriction: This type of comment cannot be nested.

Tip: These comments can contain semicolons.

Tip: You can write these comments within statements or anywhere a single blank can appear.

Details

You can use the comment statement anywhere in a SAS program to document the purpose of the program, explain unusual segments of the program, or describe steps in a complex program or calculation. SAS ignores text in comment statements during processing.

CAUTION:
Avoid placing the /* comment symbols in columns 1 and 2. In some operating environments, SAS may interpret a /* in columns 1 and 2 as a request to end the

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

Examples

These examples illustrate the two types of comments:

- This example uses the **message* format:

```
*This code finds the number in the BY group;
```

- This example uses the **message* format:

```
*----------------------------------------*
|  This uses one comment statement       |
|           to draw a box.               |
*----------------------------------------*;
```

- This example uses the /**message*/ format:

```
input @1 name $20.  /* last name    */
      @200 test 8.  /* score test   */
      @50 age 3.;   /* customer age */
```

CONTINUE

Stops processing the current DO-loop iteration and resumes with the next iteration

Valid: in a DATA step

Category: Control

Type: Executable

Restriction: Can be used only in a DO loop

Syntax

CONTINUE;

Without Arguments

The CONTINUE statement has no arguments. It stops processing statements within the current DO-loop iteration based on a condition. Processing resumes with the next iteration of the DO loop.

Comparisons

- The CONTINUE statement stops the processing of the current iteration of a loop and resumes with the next iteration; the LEAVE statement causes processing of the current loop to end.
- You can use the CONTINUE statement only in a DO loop; you can use the LEAVE statement in a DO loop or a SELECT group.

Examples

This DATA step creates a report of benefits for new full-time employees. If an employee's status is PT (part-time), the CONTINUE statement prevents the second INPUT statement and the OUTPUT statement from executing.

```
data new_emp;
   drop i;
   do i=1 to 5;
      input name $ idno status $;
         /* return to top of loop  */
         /* when condition is true */
      if status='PT' then continue;
         input benefits $10.;
         output;
   end;
   datalines;
Jones 9011 PT
Thomas 876 PT
Richards 1002 FT
Eye/Dental
Kelly 85111 PT
Smith 433 FT
HMO
;
```

See Also

Statements:

"DO, Iterative" on page 790
"LEAVE" on page 909

DATA

Begins a DATA step and provides names for any output SAS data sets

Valid: in a DATA step
Category: File-handling
Type: Declarative

Syntax

❶ **DATA** <*data-set-name-1* <(*data-set-options-1*)>>
<. . .*data-set-name-n* <(*data-set-options-n*)>>;

❷ **DATA** _NULL_;

❸ **DATA** *view-name* <*data-set-name-1* <(*data-set-options-1*)>>
<. . .*data-set-name-n* <(*data-set-options-n*)>> /
VIEW=*view-name* <(<*password-option*><SOURCE=*source-option*>)>;

❹ **DATA** *data-set-name* / PGM=*program-name*
<(<*password-option*><SOURCE=*source-option*>)>;

❺**DATA** VIEW=*view-name* <(*password-option*)>;
DESCRIBE;

❻**DATA** PGM=*program-name* <(*password-option*)>;
<**DESCRIBE**;>
<**REDIRECT** INPUT | OUTPUT *old-name-1* = *new-name-1*<. . . *old-name-n* = *new-name-n*>;>
<**EXECUTE**;>

Without Arguments

If you omit the arguments, the DATA step automatically names each successive data set that you create as DATA*n*, where *n* is the smallest integer that makes the name unique.

Arguments

data-set-name
names the SAS data file or DATA step view that the DATA step creates. To create a DATA step view, you must specify at least one *data-set-name* and that *data-set-name* must match *view-name*.

Restriction: *data-set-name* must conform to the rules for SAS names, and additional restrictions may be imposed by your operating environment.

Tip: You can execute a DATA step without creating a SAS data set. See Example 5 on page 780 for an example. For more information, see "❷When Not Creating a Data Set" on page 777.

See also: For details on the types of SAS data set names and when to use each type, see *SAS Language Reference: Concepts*.

(*data-set-options*)
specifies optional arguments that the DATA step applies when it writes observations to the output data set.

See also: Chapter 2, "Data Set Options," on page 5 for more information.

Featured in: Example 1 on page 779

NULL
specifies that SAS does not create a data set when it executes the DATA step.

VIEW=*view-name*
names a view that the DATA step uses to store the input DATA step view.

Restriction: *view-name* must match one of the data set names.

Restriction: SAS creates only one view in a DATA step.

Tip: If you specify additional data sets in the DATA statement, SAS creates these data sets when the view is processed in a subsequent DATA or PROC step. Views have the capability of generating other data sets at the time the view is executed.

Tip: SAS macro variables resolve when the view is created. Use the SYMGET function to delay macro variable resolution until the view is processed.

Featured in: Example 2 on page 779 and Example 3 on page 779

password-option
assigns a password to a stored compiled DATA step program or a DATA step view. The following password options are available:

ALTER=*alter-password*
assigns an *alter* password to a SAS data file. The password allows you to protect or replace a stored compiled DATA step program or a DATA step view.

Requirement: If you use an ALTER password in creating a stored compiled DATA step program or a DATA step view, an ALTER password is required to replace the program or view.

Requirement: If you use an ALTER password in creating a stored compiled DATA step program or a DATA step view, an ALTER password is required to execute a DESCRIBE statement.

Alias: PROTECT=

READ=*read-password*
assigns a *read* password to a SAS data file. The password allows you to read or execute a stored compiled DATA step program or a DATA step view.

Requirement: If you use a READ password in creating a stored compiled DATA step program or a DATA step view, a READ password is required to execute the program or view.

Requirement: If you use a READ password in creating a stored compiled DATA step program or a DATA step view, a READ password is required to execute DESCRIBE and EXECUTE statements. If you use an invalid password, SAS will execute the DESCRIBE statement.

Tip: If you use a READ password in creating a stored compiled DATA step program or a DATA step view, no password is required to replace the program or view.

Alias: EXECUTE=

PW=*password*
assigns a READ and ALTER password, both having the same value.

SOURCE=*source-option*
specifes one of the following source options:

SAVE
saves the source code that created a stored compiled DATA step program or a DATA step view.

ENCRYPT
encrypts and saves the source code that created a stored compiled DATA step program or a DATA step view.

Tip: If you encrypt source code, use the ALTER password option as well. SAS issues a warning message if you do not use ALTER.

NOSAVE
does not save the source code.

Default: SAVE

PGM=*program-name*
names the stored compiled program that SAS creates or executes in the DATA step. To *create* a stored compiled program, specify a slash (/) before the PGM= option. To *execute* a stored compiled program, specify the PGM= option without a slash (/).

Tip: SAS macro variables resolve when the stored program is created. Use the SYMGET function to delay macro variable resolution until the view is processed.

Featured in: Example 4 on page 779

Details

Using Both a READ and an ALTER Password If you use both a READ and an ALTER password in creating a stored compiled DATA step program or a DATA step view, the following items apply:

- A READ or ALTER password is required to execute the stored compiled DATA step program or DATA step view.
- A READ or ALTER password is required if the stored compiled DATA step program or DATA step view contains both DESCRIBE and EXECUTE statements.
  - If you use an ALTER password with the DESCRIBE and EXECUTE statements, the following items apply:
    - SAS executes both the DESCRIBE and the EXECUTE statements.
    - If you execute a stored compiled DATA step program or DATA step view with an invalid ALTER password:
      - The DESCRIBE statement does not execute.
      - In batch mode, the EXECUTE statement has no effect.
      - In interactive mode, SAS prompts you for a READ password. If the READ password is valid, SAS processes the EXECUTE statement. If it is invalid, SAS does not process the EXECUTE statement.
  - If you use a READ password with the DESCRIBE and EXECUTE statements, the following items apply:
    - In interactive mode, SAS prompts you for the ALTER password:
      - If you enter a valid ALTER password, SAS executes both the DESCRIBE and the EXECUTE statements.
      - If you enter an invalid ALTER password, SAS processes the EXECUTE statement but not the DESCRIBE statement.
    - In batch mode, SAS processes the EXECUTE statement but not the DESCRIBE statement.
    - In both interactive and batch modes, if you specify an invalid READ password SAS does not process the EXECUTE statement.
- An ALTER password is required if the stored compiled DATA step program or DATA step view contains a DESCRIBE statement.
- An ALTER password is required to replace the stored compiled DATA step program or DATA step view.

❶Creating an Output Data Set Use the DATA statement to create one or more output data sets. You can use data set options to customize the output data set. The following DATA step creates two output data sets, example1 and example2. It uses the data set option DROP to prevent the variable IDnumber from being written to the example2 data set.

```
data example1 example2 (drop=IDnumber);
   set sample;
   . . .more SAS statements. . .
run;
```

❷When Not Creating a Data Set Usually, the DATA statement specifies at least one data set name that SAS uses to create an output data set. However, when the purpose of a DATA step is to write a report or to write data to an external file, you may not want to create an output data set. Using the keyword _NULL_ as the data set name causes SAS to execute the DATA step without writing observations to a data set. This

example writes to the SAS log the value of Name for each observation. SAS does not create an output data set.

```
data _NULL_;
   set sample;
   put Name ID;
run;
```

❸Creating a DATA Step View You can create DATA step views and execute them at a later time. The following DATA step example creates a DATA step view. It uses the SOURCE=ENCRYPT option to both save and encrypt the source code.

```
data phone_list / view=phone_list (source=encrypt);
   set customer_list;
   . . .more SAS statements. . .
run;
```

For more information about DATA step views, see *SAS Language Reference: Concepts*.

❹Creating a Stored Compiled DATA Step Program The ability to compile and store DATA step programs allows you to execute the stored programs later. This can reduce processing costs by eliminating the need to compile DATA step programs repeatedly. The following DATA step example compiles and stores a DATA step program. It uses the ALTER password option, which allows the user to replace an existing stored program, and to protect the stored compiled program from being replaced.

```
data testfile / pgm=stored.test_program (alter=sales);
   set sales_data;
   . . .more SAS statements. . .
run;
```

For more information about stored compiled DATA step programs, see *SAS Language Reference: Concepts*.

❺Describing a DATA Step View The following example uses the DESCRIBE statement in a DATA step view to write a copy of the source code to the SAS log.

```
data view=inventory;
   describe;
run;
```

For information about the DESCRIBE statement, see “DESCRIBE” on page 785.

❻Executing a Stored Compiled DATA Step Program The following example executes a stored compiled DATA step program. It uses the DESCRIBE statement to write a copy of the source code to the SAS log.

```
libname stored 'SAS data library';

data pgm=stored.employee_list;
   describe;
   execute;
run;
```

For information about the DESCRIBE statement, see “DESCRIBE” on page 785. For information about the EXECUTE statement, see “EXECUTE” on page 801.

Examples

Example 1: Creating Multiple Data Files and Using Data Set Options This DATA statement creates more than one data set, and it changes the contents of the output data sets:

```
data error (keep=subject date weight)
     fitness(label='Exercise Study'
             rename=(weight=pounds));
```

The ERROR data set contains three variables. SAS assigns a label to the FITNESS data set and renames the variable *weight* to *pounds*.

Example 2: Creating Input DATA Step Views This DATA step creates an input DATA step view instead of a SAS data file:

```
libname ourlib 'SAS-data-library';

data ourlib.test / view=ourlib.test;
   set ourlib.fittest;
   tot=sum(of score1-score10);
run;
```

Example 3: Creating a View and a Data File This DATA step creates an input DATA step view named THEIRLIB.TEST and an additional temporary SAS data set named SCORETOT:

```
libname ourlib 'SAS-data-library-1';
libname theirlib 'SAS-data-library-2';

data theirlib.test scoretot
   / view=theirlib.test;
   set ourlib.fittest;
   tot=sum(of score1-score10);
run;
```

SAS does not create the data file SCORETOT until a subsequent DATA or PROC step processes the view THEIRLIB.TEST.

Example 4: Storing and Executing a Compiled Program The first DATA step produces a stored compiled program named STORED.SALESFIG:

```
libname in 'SAS-data-library-1 ';
libname stored 'SAS-data-library-2 ';

data salesdata / pgm=stored.salesfig;
   set in.sales;
   qtr1tot=jan+feb+mar;
run;
```

SAS creates the data set SALESDATA when it executes the stored compiled program STORED.SALESFIG.

```
data pgm=stored.salesfig;
run;
```

Example 5: Creating a Custom Report The second DATA step in this program produces a custom report and uses the _NULL_ keyword to execute the DATA step without creating a SAS data set:

```
data sales;
   input dept : $10. jan feb mar;
   datalines;
shoes 4344 3555 2666
housewares 3777 4888 7999
appliances 53111 7122 41333
;

data _null_;
   set sales;
   qtr1tot=jan+feb+mar;
   put 'Total Quarterly Sales: '
       qtr1tot dollar12.;
run;
```

Example 6: Using a Password With a Stored Compiled DATA Step Program The first DATA step creates a stored compiled DATA step program called STORED.ITEMS. This program includes the ALTER password, which limits access to the program.

```
libname stored 'SAS-data-library';


data employees / pgm=stored.items (alter=klondike);
   set sample;
   if TotalItems > 200 then output;
   run;
```

This DATA step executes the stored compiled DATA step program STORED.ITEMS. It uses the DESCRIBE statement to print the source code to the SAS log. Because the program was created with the ALTER password, you must use the password if you use the DESCRIBE statement. If you do not enter the password, SAS will prompt you for it.

```
data pgm=stored.items (alter=klondike);
   describe;
   execute;
run;
```

See Also

Statements:

"DESCRIBE" on page 785

"EXECUTE" on page 801

Chapter 2, "Data Set Options," on page 5

DATALINES

Indicates that data lines follow

Valid: in a DATA step

Category: File-handling

Type: Declarative

Aliases: CARDS, LINES

Restriction: Data lines cannot contain semicolons. Use "DATALINES4" on page 782 when your data contain semicolons.

Syntax

DATALINES;

Without Arguments

Use the DATALINES statement with an INPUT statement to read data that you enter directly in the program, rather than data stored in an external file.

Details

Using the DATALINES Statement The DATALINES statement is the last statement in the DATA step and immediately precedes the first data line. Use a null statement (a single semicolon) to indicate the end of the input data.

You can use only one DATALINES statement in a DATA step. Use separate DATA steps to enter multiple sets of data.

Reading Long Data Lines SAS handles data line length with the CARDIMAGE system option. If you use CARDIMAGE, SAS processes data lines exactly like 80–byte punched card images padded with blanks. If you use NOCARDIMAGE, SAS processes data lines longer than 80 columns in their entirety. Refer to "CARDIMAGE" on page 1068 for details.

Using Input Options with In-stream Data The DATALINES statement does not provide input options for reading data. However, you can access some options by using the DATALINES statement in conjunction with an INFILE statement. Specify DATALINES in the INFILE statement to indicate the source of the data and then use the options you need. See Example 2 on page 782.

Comparisons

- Use the DATALINES statement whenever data do not contain semicolons. If your data contain semicolons, use the DATALINES4 statement.
- The following SAS statements also read data or point to a location where data are stored:
  - The INFILE statement points to raw data lines stored in another file. The INPUT statement reads those data lines.
  - The %INCLUDE statement brings SAS program statements or data lines stored in SAS files or external files into the current program.
  - The SET, MERGE, MODIFY, and UPDATE statements read observations from existing SAS data sets.

Examples

Example 1: Using the DATALINES Statement In this example, SAS reads a data line and assigns values to two character variables, NAME and DEPT, for each observation in the DATA step:

```
data person;
   input name $ dept $;
   datalines;
John Sales
Mary Acctng
;
```

Example 2: Reading In-stream Data with Options This example takes advantage of options available with the INFILE statement to read in-stream data lines. With the DELIMITER= option, you can use list input to read data values that are delimited by commas instead of blanks.

```
data person;
   infile datalines delimiter=',';
   input name $ dept $;
   datalines;
John,Sales
Mary,Acctng
;
```

See Also

Statements:

"DATALINES4" on page 782
"INFILE" on page 857

System Option:

"CARDIMAGE" on page 1068

DATALINES4

Indicates that data lines that contain semicolons follow

Valid: in a DATA step
Category: File-handling
Type: Declarative
Aliases: CARDS4, LINES4

Syntax

DATALINES4;

Without Arguments

Use the DATALINES4 statement together with an INPUT statement to read data that contain semicolons that you enter directly in the program.

Details

The DATALINES4 statement is the last statement in the DATA step and immediately precedes the first data line. Follow the data lines with four consecutive semicolons that are located in columns 1 through 4.

Comparisons

Use the DATALINES4 statement when data contain semicolons. If your data do not contain semicolons, use the DATALINES statement.

Examples

In this example, SAS reads data lines that contain internal semicolons until it encounters a line of four semicolons. Execution continues with the rest of the program.

```
data biblio;
   input number citation $50.;
   datalines4;
  KIRK, 1988
2  LIN ET AL., 1995; BRADY, 1993
3  BERG, 1990; ROA, 1994; WILLIAMS, 1992
;;;;
```

See Also

Statements:

"DATALINES" on page 781

DELETE

Stops processing the current observation

Valid: in a DATA step
Category: Action

Type: Executable

Syntax

DELETE;

Without Arguments

When DELETE executes, the current observation is not written to a data set, and SAS returns immediately to the beginning of the DATA step for the next iteration.

Details

The DELETE statement is often used in a THEN clause of an IF-THEN statement or as part of a conditionally executed DO group.

Comparisons

- Use the DELETE statement when it is easier to specify a condition that excludes observations from the data set or when there is no need to continue processing the DATA step statements for the current observation.
- Use the subsetting IF statement when it is easier to specify a condition for including observations.
- Do not confuse the DROP statement with the DELETE statement. The DROP statement excludes variables from an output data set; the DELETE statement excludes observations.

Examples

Example 1: Using the DELETE Statement as Part of an IF-THEN Statement

When the value of LEAFWT is missing, the current observation is deleted:

```
if leafwt=. then delete;
```

Example 2: Using the DELETE Statement to Subset Raw Data

```
data topsales;
   infile file-specification;
   input region office product yrsales;
   if yrsales<100000 then delete;
run;
```

See Also

Statements:

"DO" on page 789
"DROP" on page 796
"IF, Subsetting" on page 847
"IF-THEN/ELSE" on page 849

DESCRIBE

Retrieves source code from a stored compiled DATA step program or a DATA step view

Valid: in a DATA step
Category: Action
Type: Executable
Restriction: Use DESCRIBE only with stored compiled DATA step programs and DATA step views.
Requirement: You must specify the PGM= or the VIEW= option in the DATA statement.

Syntax

DESCRIBE;

Without Arguments

Use the DESCRIBE statement to retrieve program source code from a stored compiled DATA step program or a DATA step view. SAS writes the source statements to the SAS log.

Details

Use the DESCRIBE statement without the EXECUTE statement to retrieve source code from a stored compiled DATA step program or a DATA step view. Use the DESCRIBE statement with the EXECUTE statement to retrieve source code and execute a stored compiled DATA step program. For information about how to use these statements with the DATA statement, see "DATA" on page 774.

See Also

Statements:

"DATA" on page 774
"EXECUTE" on page 801

DISPLAY

Displays a window that is created with the WINDOW statement

Valid: in a DATA step
Category: Window Display
Type: Executable

Syntax

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

Arguments

***window*< *.group*>**
names the window and group of fields to be displayed. This field is preceded by a period (.).

Tip: If the window has more than one group of fields, give the complete *window.group* specification; if a window contains a single unnamed group, use only *window*.

NOINPUT
specifies that you cannot input values into fields that are displayed in the window.

Default: If you omit NOINPUT, you can input values into unprotected fields that are displayed in the window.

Restriction: If you use NOINPUT in all DISPLAY statements in a DATA step, you *must* include a STOP statement to stop processing the DATA step.

Tip: The NOINPUT option is useful when you want to allow values to be entered into a window at some times but not others. For example, you can display a window once for entering values and a second time for verifying them.

BLANK
clears the window.

Tip: Use the BLANK option when you want to display different groups of fields in a window and you do not want text from the previous group to appear in the current display.

BELL
produces an audible alarm, beep, or bell sound when the window is displayed if your terminal is equipped with a speaker device that provides sound.

DELETE
deletes the display of the window after processing passes from the DISPLAY statement on which the option appears.

Details

You must create a window in the same DATA step that you use to display it. Once you display a window, the window remains visible until you display another window over it or until the end of the DATA step. When you display a window that contains fields where you enter values, either enter a value or press ENTER at *each* unprotected field to cause SAS to proceed to the next display. You cannot skip any fields.

While a window is being displayed, use commands and function keys to view other windows, to change the size of the current window, and so on.

A DATA step that contains a DISPLAY statement continues execution until the last observation that is read by a SET, MERGE, UPDATE, MODIFY, or INPUT statement

has been processed or until a STOP or ABORT statement is executed. You can also issue the END command on the command line of the window to stop the execution of the DATA step.

You must create a window before you can display it. See the WINDOW statement later in this chapter for a description of how to create windows. A window that is displayed with the DISPLAY statement does not become part of the SAS log or output file.

Examples

This DATA step creates and displays a window named START. The START window fills the entire screen. Both lines of text are centered.

```
data _null_;
   window start
          #5 @28 'WELCOME TO THE SAS SYSTEM'
          #12 @30 'PRESS ENTER TO CONTINUE';
   display start;
   stop;
run;
```

Although the START window in this example does not require you to input any values, you must press ENTER to cause SAS execution to proceed to the STOP statement. If you omit the STOP statement, the DATA step executes endlessly unless you enter END on the command line of the window.

Note: Because this DATA step does not read any observations, SAS cannot detect an end-of-file to cause DATA step execution to cease. If you add the NOINPUT option to the DISPLAY statement, the window displays quickly and is removed. △

See Also

Statement:

"WINDOW" on page 1033

DM

Submits SAS Program Editor, Log, Procedure Output or text editor commands as SAS statements

Valid: anywhere

Category: Program Control

Syntax

DM <*window*> *'command(s)'* <*window*> <CONTINUE>;

Arguments

window

specifies the active window. For more information, see .

Default: If you omit the window name, SAS uses the Program Editor window as the default.

'command'
can be any windowing command or text editor command and must be enclosed in single quotation marks. If you want to issue several commands, separate them with semicolons.

CONTINUE
causes SAS to execute any SAS statements that follow the DM statement in the Program Editor window and, if a windowing command in the DM statement called a window, makes that window active.

Tip: Any windows that are activated by the SAS statements (such as the Output window) appear before the window that is to be made active.

Note: If you specify Log as the active window, for example, and have other SAS statements that follow the DM statement (for example, in an autoexec file), those statements are not submitted to SAS until control returns to the SAS interface.

Details

Execution occurs when the DM statement is submitted to SAS. This statement is useful for

- changing SAS interface features during a SAS session
- changing SAS interface features at the beginning of each SAS session by placing the DM statement in an autoexec file
- performing utility functions in windowing applications, such as saving a file with the FILE command or clearing a window with the CLEAR command.

Window placement affects the outcome of the statement:

- If you name a window before the commands, those commands apply to that window.
- If you name a window after the commands, SAS executes the commands and then makes that window the active window. The active window is opened and contains the cursor.

Examples

Example 1: Using the DM Statement

- `dm 'color text cyan; color command red';`
- ```
  dm log 'clear; pgm; color numbers green'
     output;
  ```
- `dm 'caps on';`
- `dm log 'clear' output;`

Example 2: Using the CONTINUE Option with SAS Statements That Do Not Activate a Window

This example causes SAS to display the first window of the SAS/AF application, executes the DATA step, moves the cursor to the first field of the SAS/AF application window, and makes that window active.

```
dm 'af c=your-program' continue;
```
```

```
data temp;
   . . . more SAS statements . . .
run;
```

Example 3: Using the CONTINUE Option with SAS Statements That Activate a Window This example displays the first window of the SAS/AF application and executes the PROC PRINT step, which activates the OUTPUT window. Closing the OUTPUT window moves the cursor to the last active window..

```
dm 'af c=your-program' continue;

proc print data=temp;
run;
```

DO

Designates a group of statements to be executed as a unit

Valid: in a DATA step
Category: Control
Type: Executable

Syntax

DO;
   ...*more SAS statements*...
END;

Without Arguments

Use the DO statement for simple DO group processing.

Details

The DO statement is the simplest form of DO group processing. The statements between the DO and END statements are called a *DO group*. You can nest DO statements within DO groups.

Note: The memory capabilities of your system may limit the number of nested DO statements you can use. For details, see the SAS documentation about how many levels of nested DO statements your system's memory can support. △

A simple DO statement is often used within IF-THEN/ELSE statements to designate a group of statements to be executed depending on whether the IF condition is true or false.

Comparisons

There are three other forms of the DO statement:

- The iterative DO statement executes statements between DO and END statements repetitively based on the value of an index variable. The iterative DO statement can contain a WHILE or UNTIL clause.

- The DO UNTIL statement executes statements in a DO loop repetitively until a condition is true, checking the condition after each iteration of the DO loop.
- The DO WHILE statement executes statements in a DO loop repetitively while a condition is true, checking the condition before each iteration of the DO loop.

Examples

In this simple DO group, the statements between DO and END are performed only when YEARS is greater than 5. If YEARS is less than or equal to 5, statements in the DO group do not execute, and the program continues with the assignment statement that follows the ELSE statement.

```
if years>5 then
   do;
      months=years*12;
      put years= months=;
   end;
   else yrsleft=5-years;
```

See Also

Statements:

"DO, Iterative" on page 790
"DO UNTIL" on page 794
"DO WHILE" on page 795

DO, Iterative

Executes statements between DO and END repetitively based on the value of an index variable

Valid: in a DATA step
Category: Control
Type: Executable

Syntax

DO *index-variable=specification-1 <, . . . specification-n>*;
. . . more SAS statements . . .
END;

Arguments

index-variable
names a variable whose value governs execution of the DO group. The *index-variable* argument is required.

Tip: Unless you specify to drop it, the index variable is included in the data set that is being created.

CAUTION:

Avoid changing the index variable within the DO group. If you modify the index variable within the iterative DO group, you may cause infinite looping. △

specification

denotes an expression or a series of expressions in this form

start <TO *stop*> <BY *increment*>
<WHILE(*expression*) | UNTIL(*expression*)>

Requirement: The interative DO statement requires at least one *specification* argument.

Tip: The order of the optional TO and BY clauses can be reversed.

Tip: When you use more than one *specification*, each one is evaluated prior to its execution.

start

specifies the initial value of the index variable.

Restriction: When it is used with TO *stop* or BY *increment*, *start* must be a number or an expression that yields a number.

Explanation: When it is used without TO *stop* or BY *increment*, the value of *start* can be a series of items expressed in this form:

item-1 <, . . . item-n >;

The items may be either all numeric or all character constants, or they may be variables. Enclose character constants in quotation marks. The DO group is executed once for each value in the list. If a WHILE condition is added, it applies only to the item that it immediately follows.

The DO group is executed first with *index-variable* equal to *start*. The value of *start* is evaluated before the first execution of the loop.

Featured in: Example 1 on page 792

TO *stop*

optionally specifies the ending value of the index variable.

Restriction: *Stop* must be a number or an expression that yields a number.

Explanation: When both *start* and *stop* are present, execution continues (based on the value of *increment*) until the value of *index-variable* passes the value of *stop*. When only *start* and *increment* are present, execution continues (based on the value of *increment*) until a statement directs execution out of the loop, or until a WHILE or UNTIL expression that is specified in the DO statement is satisfied. If neither *stop* nor *increment* is specified, the group executes according to the value of *start*. The value of *stop* is evaluated before the first execution of the loop.

Tip: Any changes to *stop* made within the DO group do not affect the number of iterations. To stop iteration of a loop before it finishes processing, change the value of *index-variable* so that it passes the value of *stop*, or use a LEAVE statement to go to a statement outside the loop.

Featured in: Example 1 on page 792

BY *increment*

optionally specifies a positive or negative number (or an expression that yields a number) to control the incrementing of *index-variable*.

Explanation: The value of *increment* is evaluated prior to the execution of the loop. Any changes to the increment that are made within the DO group do not affect the number of iterations. If no increment is specified, the index variable is increased by 1. When *increment* is positive, *start* must be the lower bound and

stop, if present, must be the upper bound for the loop. If *increment* is negative, *start* must be the upper bound and *stop*, if present, must be the lower bound for the loop.

Featured in: Example 1 on page 792

WHILE(*expression*) UNTIL(*expression*)
optionally evaluates, either before or after execution of the DO group, any SAS expression that you specify. Enclose the expression in parentheses.

Restriction: A WHILE or UNTIL specification affects only the last item in the clause in which it is located.

Explanation: A WHILE expression is evaluated before each execution of the loop, so that the statements inside the group are executed repetitively while the expression is true. An UNTIL expression is evaluated after each execution of the loop, so that the statements inside the group are executed repetitively until the expression is true.

Featured in: Example 1 on page 792

See Also: "DO WHILE" on page 795 and "DO UNTIL" on page 794 for more information.

Comparisons

There are three other forms of the DO statement:

- The DO statement, the simplest form of DO-group processing, designates a group of statements to be executed as a unit, usually as a part of IF-THEN/ELSE statements.
- The DO UNTIL statement executes statements in a DO loop repetitively until a condition is true, checking the condition after each iteration of the DO loop.
- The DO WHILE statement executes statements in a DO loop repetitively while a condition is true, checking the condition before each iteration of the DO loop.

Examples

Example 1: Using Various Forms of the Iterative DO Statement

- These iterative DO statements use a list of items for the value of *start*:
  - `do month='JAN','FEB','MAR';`
  - `do count=2,3,5,7,11,13,17;`
  - `do i=5;`
  - `do i=var1--var5;`
  - `do i=var1, var2, var3;`
  - `do i='01JAN2001'd,'25FEB2001'd,'18APR2001'd;`

- These iterative DO statements use the *start* TO *stop* syntax:
  - `do i=1 to 10;`
  - `do i=1 to exit;`
  - `do i=1 to x-5;`
  - `do i=1 to k-1, k+1 to n;`
  - `do i=k+1 to n-1;`

- These iterative DO statements use the BY *increment* syntax:
  - `do i=n to 1 by -1;`
  - ```
    do i=.1 to .9 by .1, 1 to 10 by 1,
       20 to 100 by 10;
    ```
  - `do count=2 to 8 by 2;`
- These iterative DO statements use WHILE and UNTIL clauses:
  - `do i=1 to 10 while(x<y);`
  - `do i=2 to 20 by 2 until((x/3)>y);`
  - `do i=10 to 0 by -1 while(month='JAN');`
- In this example, the DO loop is executed when I=1 and I=2; the WHILE condition is evaluated when I=3, and the DO loop is executed if the WHILE condition is true.

  ```
  DO I=1,2,3 WHILE (condition);
  ```

Example 2: Using the Iterative DO Statement without Infinite Looping In each of the following examples, the DO group is executed ten times. The first example demonstrates the preferred approach.

```
   /* correct coding */
do i=1 to 10;
   ...more SAS statements...
end;
```

In the next example, if you do not specify a WHILE or UNTIL clause in the DO statement, infinite looping can occur.

```
   /* Warning --- infinite looping can occur */
do i=1 to n by m;
   ...more SAS statements...
   if i=10 then leave;
end;
if i=10 then put 'EXITED LOOP';
```

This example uses an UNTIL clause to set a flag; then it checks the flag during each iteration of the loop.

```
flag=0;
do i=1 to 10 until(flag);
   ...more SAS statements...
   if expression then flag=1;
   ...more SAS statements...
end;
```

Example 3: Stopping Execution of the DO Loop

- In this example, setting the value of the index variable to the current value of EXIT causes the loop to terminate.

  ```
  data iterate1;
     input x;
     exit=10;
     do i=1 to exit;
        y=x*normal(0);
           /* if y>25,           */
           /* changing i's value */
  ```
  ```

```
            /* stops execution    */
         if y>25 then i=exit;
         output;
      end;
      datalines;
5
000
2500
;
```

See Also

Statements:

"ARRAY" on page 755
"Array Reference" on page 759
"DO" on page 789
"DO UNTIL" on page 794
"DO WHILE" on page 795
"GO TO" on page 845

DO UNTIL

Executes statements in a DO loop repetitively until a condition is true

Valid: in a DATA step
Category: Control
Type: Executable

Syntax

DO UNTIL (*expression*);
...more SAS statements...

END;

Arguments

(expression)
is any SAS expression, enclosed in parentheses. You must specify at least one *expression*.

Details

The expression is evaluated at the bottom of the loop after the statements in the DO loop have been executed. If the expression is true, the DO loop does not iterate again.

Note: The DO loop always iterates at least once. △

Comparisons

There are three other forms of the DO statement:

- The DO statement, the simplest form of DO-group processing, designates a group of statements to be executed as a unit, usually as a part of IF-THEN/ELSE statements.
- The iterative DO statement executes statements between DO and END statements repetitively based on the value of an index variable.
- The DO WHILE statement executes statements in a DO loop repetitively while a condition is true, checking the condition before each iteration of the DO loop. The DO UNTIL statement evaluates the condition at the bottom of the loop; the DO WHILE statement evaluates the condition at the top of the loop.

  Note: The statements in a DO UNTIL loop always execute at least one time, whereas the statements in a DO WHILE loop do not iterate even once if the condition is false. △

Examples

These statements repeat the loop until N is greater than or equal to 5. The expression N>=5 is evaluated at the bottom of the loop. There are five iterations in all (0, 1, 2, 3, 4).

```
n=0;
   do until(n>=5);
      put n=;
      n+1;
   end;
```

See Also

Statements:

"DO" on page 789
"DO, Iterative" on page 790
"DO WHILE" on page 795

DO WHILE

Executes statements repetitively while a condition is true

Valid: in a DATA step
Category: Control
Type: Executable

Syntax

DO WHILE (*expression*);
...more SAS statements...
END;

Arguments

(*expression*)
is any SAS expression, enclosed in parentheses. You must specify at least one *expression*.

Details

The expression is evaluated at the top of the loop before the statements in the DO loop are executed. If the expression is true, the DO loop iterates. If the expression is false the first time it is evaluated, the DO loop does not iterate even once.

Comparisons

There are three other forms of the DO statement:

- The DO statement, the simplest form of DO-group processing, designates a group of statements to be executed as a unit, usually as a part of IF-THEN/ELSE statements.
- The iterative DO statement executes statements between DO and END statements repetitively based on the value of an index variable.
- The DO UNTIL statement executes statements in a DO loop repetitively until a condition is true, checking the condition after each iteration of the DO loop. The DO WHILE statement evaluates the condition at the top of the loop; the DO UNTIL statement evaluates the condition at the bottom of the loop.

  Note: If the expression is false, the statements in a DO WHILE loop do not execute. However, because the DO UNTIL expression is evaluated at the bottom of the loop, the statements in the DO UNTIL loop always execute at least once. △

Examples

These statements repeat the loop while N is less than 5. The expression N<5 is evaluated at the top of the loop. There are five iterations in all (0, 1, 2, 3, 4).

```
n=0;
   do while(n<5);
      put n=;
      n+1;
   end;
```

See Also

Statements:

"DO" on page 789
"DO, Iterative" on page 790
"DO UNTIL" on page 794

DROP

Excludes variables from output SAS data sets

Valid: in a DATA step
Category: Information
Type: Declarative

Syntax

DROP *variable-list*;

Arguments

variable-list
specifies the names of the variables to omit from the output data set.

Tip: You can list the variables in any form that SAS allows.

Details

The DROP statement applies to all the SAS data sets that are created within the same DATA step and can appear anywhere in the step. The variables in the DROP statement are available for processing in the DATA step. If no DROP or KEEP statement appears, all data sets that are created in the DATA step contain all variables. Do not use both DROP and KEEP statements within the same DATA step.

Comparisons

- The DROP statement differs from the DROP= data set option in the following ways:
  - You cannot use the DROP statement in SAS procedure steps.
  - The DROP statement applies to all output data sets that are named in the DATA statement. To exclude variables from some data sets but not from others, use the DROP= data set option in the DATA statement.
- The KEEP statement is a parallel statement that specifies a list of variables to write to output data sets. Use the KEEP statement instead of the DROP statement if the number of variables to include is significantly smaller than the number to omit.
- Do not confuse the DROP statement with the DELETE statement. The DROP statement excludes variables from output data sets; the DELETE statement excludes observations.

Examples

- These examples show the correct syntax for listing variables with the DROP statement:
  - `drop time shift batchnum;`
  - `drop grade1-grade20;`
- In this example, the variables PURCHASE and REPAIR are used in processing but are not written to the output data set INVENTRY:

```
data inventry;
   drop purchase repair;
   infile file-specification;
   input unit part purchase repair;
   totcost=sum(purchase,repair);
run;
```

See Also

Data Set Option:

"DROP=" on page 15

Statements:

"DELETE" on page 783

"KEEP" on page 905

END

Ends a DO group or a SELECT group

Valid: in a DATA step

Category: Control

Type: Declarative

Syntax

END;

Without Arguments

Use the END statement to end DO group or SELECT group processing.

Details

The END statement must be the last statement in a DO group or a SELECT group.

Examples

This example shows a simple DO group and a simple SELECT group:

```
□ do;
        . . .more SAS statements. . .
  end;

□ select(expression);
     when(expression) SAS statement;
     otherwise SAS statement;
  end;
```

See Also

Statements:

"DO" on page 789

"SELECT" on page 1007

ENDSAS

Terminates a SAS job or session after the current DATA or PROC step executes

Valid: anywhere

Category: Program Control

Syntax

ENDSAS ;

Without Arguments

The ENDSAS statement terminates a SAS job or session.

Details

ENDSAS is most useful in interactive or windowing sessions.

Note: ENDSAS statements cannot be part of other statements such as IF-THEN statements. △

Comparisons

You can also terminate a SAS job or session by using the BYE or the ENDSAS command from any SAS window command line. For details, refer to the online help for SAS windows.

See Also

The macro variable SYSSTARTID in *SAS Macro Language: Reference*

ERROR

Sets _ERROR_ to 1 and, optionally, writes a message to the SAS log

Valid: in a DATA step

Category: Action

Type: Executable

Syntax

ERROR <*message*>;

Without Arguments

Using ERROR without an argument sets the automatic variable _ERROR_ to 1 without printing any message in the log.

Arguments

message
writes a message to the log.

Tip: *Message* can include character literals (enclosed in quotation marks), variable names, formats, and pointer controls.

Details

The ERROR statement sets the automatic variable _ERROR_ to 1 and, optionally, writes a message that you specify to the SAS log. When _ERROR_ = 1, SAS writes the data lines that correspond to the current observation in the SAS log.

Using ERROR is equivalent to using these statements in combination:

- an assignment statement setting _ERROR_ to 1
- a FILE LOG statement
- a PUT statement (if you specify a message)
- another FILE statement resetting FILE to any previously specified setting.

Examples

In the following examples, SAS writes the error message and the variable name and value to the log for each observation that satisfies the condition in the IF-THEN statement.

- In this example, the ERROR statement automatically resets the FILE statement specification to the previously specified setting.

```
file file-specification;
   if type='teen' & age > 19 then
      error 'type and age don"t match ' age=;
```

- This example uses a series of statements to produce the same results.

```
file file-specification;
   if type='teen' & age > 19 then
      do;
         file log;
         put 'type and age don"t match ' age=;
         _error_=1;
         file file-specification;
      end;
```

See Also

Statement:
"PUT" on page 962

EXECUTE

Executes a stored compiled DATA step program

Valid: in a DATA step
Category: Action
Type: Executable
Restriction: Use EXECUTE with stored compiled DATA step programs only.
Requirement: You must specify the PGM= option in the DATA step.

Syntax

EXECUTE;

Without Arguments

The EXECUTE statement executes a stored compiled DATA step program.

Details

Use the DESCRIBE statement with the EXECUTE statement in the same DATA step to retrieve the source code and execute a stored compiled DATA step program. If you do not specify either statement, EXECUTE is assumed. The order in which you use the statements is interchangeable. The DATA step program executes when it reaches a step boundary. For information about how to use these statements with the DATA statement, see "DATA" on page 774.

See Also

Statements:

"DATA" on page 774
"DESCRIBE" on page 785

FILE

Specifies the current output file for PUT statements

Valid: in a DATA step
Category: File-handling
Type: Executable

Syntax

FILE *file-specification* <*options*> <*host-options*>;

Arguments

file-specification
identifies an external file that the DATA step uses to write output from a PUT statement. *File-specification* can have these forms:

'external-file'
specifies the physical name of an external file, which is enclosed in quotation marks. The physical name is the name by which the operating environment recognizes the file.

fileref
specifies the fileref of an external file.

Requirement: You must have previously associated *fileref* with an external file in a FILENAME statement or function, or in an appropriate operating environment command. There is only one exception to this rule: when you use the FILEVAR= option, the fileref is simply a placeholder.

See Also: "FILENAME" on page 821

fileref(file)
specifies a fileref that is previously assigned to an external file that is an aggregate grouping of files. Follow the fileref with the name of a file or member, which is enclosed in parentheses.

Requirement: You must previously associate *fileref* with an external file in a FILENAME statement or function, or in an appropriate operating environment command.

See Also: "FILENAME" on page 821

Operating Environment Information: Different operating environments call an aggregate grouping of files by different names, such as a directory, a MACLIB, or a partitioned data set. For details, see the SAS documentation for your operating environment. △

LOG
is a reserved fileref that directs the output that is produced by any PUT statements to the SAS log.

At the beginning of each execution of a DATA step, the fileref that indicates where the PUT statements write is automatically set to LOG. Therefore, the first PUT statement in a DATA step always writes to the SAS log, unless it is preceded by a FILE statement that specifies otherwise.

Tip: Because output lines are by default written to the SAS log, use a FILE LOG statement to restore the default action or to specify additional FILE statement options.

PRINT
is a reserved fileref that directs the output that is produced by any PUT statements to the same print file as the output that is produced by SAS procedures.

Interaction: When you write to a print file, the value of the N= option must be either 1 or PAGESIZE.

Tip: When PRINT is the fileref, SAS uses carriage control characters and writes the output with the characteristics of a print file.

See Also: A complete discussion of print files in "DATA Step Processing" in *SAS Language Reference: Concepts*.

Operating Environment Information: The carriage control characters that are written to a file can be specific to the operating environment. For details, see the SAS documentation for your operating environment. △

Options

BLKSIZE=*block-size*
specifies the block size of the output file.

Default: Depends on your operating environment.

Operating Environment Information: The default value of the block size is dependent on the operating environment. For details, see the SAS documentation for your operating environment. △

COLUMN=*variable*
specifies a variable that SAS automatically sets to the current column location of the pointer. This variable, like automatic variables, is not written to the data set.

Alias: COL=

DELIMITER= *'quoted string'* | *character_variable*
specifies an alternate delimiter (other than blank) to be used for LIST output. It is ignored for other types of output (formatted, column, named).

Alias: DLM

Restriction: Even though a character string or character variable is accepted, only the first character of the string or variable is used as the output delimiter. This differs from INFILE DELIMITER= processing.

Interaction: Output that contains embedded delimiters requires the DSD option.

Tip: The delimiter can be used with the colon (:) modifier (modified LIST output).

See Also: DSD on page 804

DROPOVER
discards data items that exceed the output line length (as specified by the LINESIZE= or LRECL= options in the FILE statement).

Default: FLOWOVER

Explanation: By default, data that exceed the current line length are written on a new line. When you specify DROPOVER, SAS drops (or ignores) an entire item when there is not enough space in the current line to write it. When this occurs, the column pointer remains positioned after the last value that is written in the current line. Thus, the PUT statement may write other items in the current output line if they fit in the space that remains or if the column pointer is repositioned. When a data item is dropped, the DATA step continues normal execution (_ERROR_=0). At the end of the DATA step, a message is printed for each file from which data were lost.

Tip: Use DROPOVER when you want the DATA step to continue executing if the PUT statement attempts to write past the current line length, but you do not want the data item that exceeds the line length to be written on a new line.

See Also: FLOWOVER on page 805 and STOPOVER on page 809

DSD
enables you to write data items that contains embedded delimiters to LIST output. It is ignored for other types of output (formatted, column, named). Any data item that contains the specified delimiter is quoted with the double quotation mark (") as it is output. Any double quotation marks that are embedded in the data item are promoted. For example, the data item, *Dad"s*, is written as *"Dad" "s"* in LIST output.

Interaction: If you specify DSD, the default delimiter is assumed to be the comma (,). Specify the DELIMITER= option if you want to use a different delimiter.

Tip: By default, data items that do not contain the specified delimiter are not quoted. However, you can use the tilde (~) modifier to force any data item to be quoted, even if it contains no embedded delimiter.

See Also: DELIMITER= on page 804

FILENAME=*variable*
defines a character variable, whose name you supply, that SAS sets to the value of the physical name of the file currently open for PUT statement output. The physical name is the name by which the operating environment recognizes the file.

Tip: This variable, like automatic variables, is not written to the data set.

Tip: Use a LENGTH statement to make the variable length long enough to contain the value of the physical filename if it is longer than eight characters (the default length of a character variable).

See Also: FILEVAR= on page 805

Featured in: Example 4 on page 813

FILEVAR=*variable*
defines a variable whose change in value causes the FILE statement to close the current output file and open a new one the next time the FILE statement executes. The next PUT statement that executes writes to the new file that is specified as the value of the FILEVAR= variable.

Restriction: The value of a FILEVAR= variable is expressed as a character string that contains a physical filename.

Interaction: When you use the FILEVAR= option, the *file-specification* is just a placeholder, not an actual filename or a fileref that has been previously-assigned to a file. SAS uses this placeholder for reporting processing information to the SAS log. It must conform to the same rules as a fileref.

Tip: This variable, like automatic variables, is not written to the data set.

Tip: If any of the physical filenames is longer than eight characters (the default length of a character variable), assign the FILEVAR= variable a longer length with another statement, such as a LENGTH statement or an INPUT statement.

See Also: FILENAME= on page 805

Featured in: Example 5 on page 814

FLOWOVER
causes data that exceed the current line length to be written on a new line. When a PUT statement attempts to write beyond the maximum allowed line length (as specified by the LINESIZE= option in the FILE statement), the current output line is written to the file and the data item that exceeds the current line length is written to a new line.

Default: FLOWOVER

Interaction: If the PUT statement contains a trailing @, the pointer is positioned after the data item on the new line, and the next PUT statement writes to that line. This process continues until the end of the input data is reached or until a PUT statement without a trailing @ causes the current line to be written to the file.

See Also: DROPOVER on page 804 and STOPOVER on page 809

FOOTNOTES | NOFOOTNOTES
controls whether currently defined footnotes are printed.

Alias: FOOTNOTE | NOFOOTNOTE

Requirement: In order to print footnotes in a DATA step report, you must set the FOOTNOTE option in the FILE statement.

Default: NOFOOTNOTES

HEADER=*label*
defines a statement label that identifies a group of SAS statements that you want to execute each time SAS begins a new output page.

Restriction: The first statement after the label must be an executable statement. Thereafter you can use any SAS statement.

Restriction: Use the HEADER= option only when you write to print files.

Tip: To prevent the statements in this group from executing with each iteration of the DATA step, use two RETURN statements: one precedes the label and the other appears as the last statement in the group.

Featured in: Example 1 on page 812

LINE=*variable*
defines a variable whose value is the current relative line number within the group of lines available to the output pointer. You supply the variable name; SAS automatically assigns the value.

Range: 1 to the value that is specified by the N= option or with the #*n* line pointer control. If neither is specified, the LINE= variable has a value of 1.

Tip: This variable, like automatic variables, is not written to the data set.

Tip: The value of the LINE= variable is set at the end of PUT statement execution to the number of the next available line.

LINESIZE=*line-size*
sets the maximum number of columns per line for reports and the maximum record length for data files.

Alias: LS=

Default: The default LINESIZE= value is determined by one of two options:

- the LINESIZE= system option when you write to a print file (a file that contains carriage control characters) or to the SAS log.
- the LRECL= option in the FILE statement when you write to a nonprint file.

Range: From 64 to the maximum logical record length that is allowed for a specific file in your operating environment. For details, see the SAS documentation for your operating environment.

Operating Environment Information: The highest value allowed for LINESIZE= is dependent on your operating environment. △

Interaction: If a PUT statement tries to write a line that is longer than the value that is specified by the LINESIZE= option, the action that is taken is determined by whether FLOWOVER, DROPOVER, or STOPOVER is in effect. By default (FLOWOVER), SAS writes the line as two or more separate records.

Comparisons: LINESIZE= tells SAS how much of the line to use. LRECL= specifies the physical record length of the file.

See Also: LRECL= on page 807, DROPOVER on page 804, FLOWOVER on page 805, and STOPOVER on page 809

Featured in: Example 6 on page 814

LINESLEFT=*variable*
defines a variable whose value is the number of lines left on the current page. You supply the variable name; SAS assigns that variable the value of the number of lines left on the current page. The value of the LINESLEFT= variable is set at the end of PUT statement execution.

Alias: LL=

Tip: This variable, like automatic variables, is not written to the data set.

Featured in: Example 2 on page 812

LRECL=*logical-record-length*
specifies the logical record length of the output file.

Default: If you omit the LRECL= option, SAS chooses a value based on the operating environment's file characteristics.

Operating Environment Information: Values for *logical-record-length* are dependent on the operating environment. For details, see the SAS documentation for your operating environment. △

Comparisons: LINESIZE= tells SAS how much of the line to use; LRECL= specifies the physical line length of the file.

See Also: LINESIZE= on page 806, PAD on page 808, and PAGESIZE= on page 809

MOD
writes the output lines after any existing lines in the file.

Default: OLD

Restriction: MOD is not accepted under all operating environments.

Operating Environment Information: For details, see the SAS documentation for your operating environment. △

See Also: OLD on page 808

N=*available-lines*
specifies the number of lines that you want available to the output pointer in the current iteration of the DATA step. *Available-lines* can be expressed as a number (*n*) or as the keyword PAGESIZE or PS.

n
specifies the number of lines that are available to the output pointer. The system can move back and forth between the number of lines that are specified while composing them before moving on to the next set.

PAGESIZE
specifies that the entire page is available to the output pointer.

Alias: PS

Restriction: N=PAGESIZE is valid only when output is sent to a print file.

Restriction: If the current output file is a print file, *available-lines* must have a value of either 1 or PAGESIZE.

Interactions: There are two ways to control the number of lines available to the output pointer:

- the N= option
- the #*n* line pointer control in a PUT statement.

Interaction: If you omit the N= option and no # pointer controls are used, one line is available; that is, by default, N=1. If N= is not used but there are # pointer controls, N= is assigned the highest value that is specified for a # pointer control in any PUT statement in the current DATA step.

Tip: Setting N=PAGESIZE enables you to compose a page of multiple columns one column at a time.

Featured in: Example 3 on page 813

NBYTE=*variable*
specifies the name of a variable that contains the number of bytes to write to a file when you write data in stream record format (RECFM=S in the FILENAME statement). By default, the number of bytes that are read is equal to the actual LRECL value of the file. NBYTE= is used with the SOCKET and FTP access methods only.

Tip: If the number of bytes to be read is set to -1, the FTP and SOCKET access methods return the number of bytes that are currently available in the input buffer.

Featured in: Example 7 on page 814

See also: the RECFM= option on page 832 in the FILENAME-FTP statement

ODS < = (*ODS suboptions*) >
specifies to use the Output Delivery System to format the output from a DATA step. It defines the structure of the data component and binds that component to a table definition to produce an output object. ODS sends this object to all open ODS destinations, each of which formats the object appropriately. For information on the *ODS suboptions*, see "ODS Suboptions" on page 817. For general information on the Output Delivery System, see*The Complete Guide to the SAS Output Delivery System*.

Default: If you omit the ODS suboptions, the DATA step uses a default table definition (base.datastep.table) that is stored in the SASHELP data library. This definition defines two generic columns: one for character variables, and one for numeric variables. ODS associates each variable in the DATA step with one of these columns and displays the variables in the order in which they are defined in the DATA step.

Without suboptions, the default table definition uses the variable's label as its column header. If no label exists, the definition uses the variable's name as the column header.

Requirement: The ODS option is valid only when you use the fileref PRINT in the FILE statement.

Restriction: You cannot use the _FILE_=, FILEVAR=, HEADER=, and PAD options with the ODS option.

Interaction: The DELIMITER= and DSD options have no effect on the ODS option. The FOOTNOTES|NOFOOTNOTES, LINESIZE, PAGESIZE, and TITLES | NOTITLES options only have an effect on the listing destination.

OLD
replaces the previous contents of the file.

Default: OLD

Restriction: OLD is not accepted under all operating environments.

Operating Environment Information: For details, see the SAS documentation for your operating environment. △

See Also: MOD on page 807

PAD | NOPAD
controls whether records written to an external file are padded with blanks to the length that is specified in the LRECL= option.

Default: NOPAD is the default when writing to a variable-length file; PAD is the default when writing to a fixed-length file.

Tip: PAD provides a quick way to create fixed-length records in a variable-length file.

See Also: LRECL= on page 807

PAGESIZE=*value*
sets the number of lines per page for your reports.

Alias: PS=

Default: the value of the PAGESIZE= system option.

Range: The value may range from 15 to 32767.

Interaction: If any TITLE statements are currently defined, the lines they occupy are included in counting the number of lines for each page.

Tip: After the value of the PAGESIZE= option is reached, the output pointer advances to line 1 of a new page.

See Also: "PAGESIZE=" on page 1133

PRINT | NOPRINT
controls whether carriage control characters are placed in the output lines.

Restriction: When you write to a print file, the value of the N= option must be either 1 or PAGESIZE.

Tip: The PRINT option is not necessary if you are using fileref PRINT.

Operating Environment Information: The carriage control characters that are written to a file can be specific to the operating environment. For details, see the SAS documentation for your operating environment. △

RECFM=*record-format*
specifies the record format of the output file.

Range: Values are dependent on the operating environment.

Operating Environment Information: Values for *record-format* are dependent on the operating environment. For details, see the SAS documentation for your operating environment. △

STOPOVER
stops processing the DATA step immediately if a PUT statement attempts to write a data item that exceeds the current line length. In such a case, SAS discards the data item that exceeds the current line length, writes the portion of the line that was built before the error occurred, and issues an error message.

Default: FLOWOVER

See Also: FLOWOVER on page 805 and DROPOVER on page 804

TITLES | NOTITLES
controls the printing of the current title lines on the pages of print files. When NOTITLES is omitted, or when TITLES is specified, SAS prints any titles that are currently defined.

Alias: TITLE | NOTITLE

Default: TITLES

FILE=*variable*
names a character variable that references the current output buffer of this FILE statement. You can use the variable in the same way as any other variable, even as the target of an assignment. The variable is automatically retained and initialized to blanks. Like automatic variables, the _FILE_= variable is not written to the data set.

Restriction: *variable* cannot be a previously defined variable. Make sure that the _FILE_= specification is the first occurrence of this variable in the DATA step. Do not set or change the length of _FILE_= variable with the LENGTH or

ATTRIB statements. However, you can attach a format to this variable with the ATTRIB or FORMAT statement.

Interaction: The maximum length of this character variable is the logical record length (LRECL) for the specified FILE statement. However, SAS does not open the file to know the LRECL until prior to the execution phase. Therefore, the designated size for this variable during the compilation phase is 32,767.

Tip: Modification of this variable directly modifies the FILE statement's current output buffer. Any subsequent PUT statement for this FILE statement outputs the contents of the modified buffer. The _FILE_= variable accesses only the current output buffer of the specified FILE statement even if you use the N= option to specify multiple output buffers.

Tip: To access the contents of the output buffer in another statement without using the _FILE_= option, use the automatic variable _FILE_.

Main Discussion: "Accessing the Contents of the Output Buffer" on page 810

Host Options

Operating Environment Information: For descriptions of host-specific options on the FILE statement, see the SAS documentation for your operating environment. △

Details

Overview By default, PUT statement output is written to the SAS log. Use the FILE statement to route this output to either the same external file to which procedure output is written or to a different external file. You can indicate whether or not carriage control characters should be added to the file. See the PRINT | NOPRINT option on page 809.

You can use the FILE statement in conditional (IF-THEN) processing because it is executable. You can also use multiple FILE statements to write to more than one external file in a single DATA step.

Operating Environment Information: Using the FILE statement requires host-specific information. See the SAS documentation for your operating environment before you use this statement. △

You can now use the Output Delivery System with the FILE statement to write DATA step results. This functionality is briefly discussed here. For details, see "FILE, ODS" on page 815. For further information, see *The Complete Guide to the SAS Output Delivery System*.

Updating an External File in Place You can use the FILE statement with the INFILE and PUT statements to update an external file in place, updating either an entire record or only selected fields within a record. Follow these guidelines:

- Always place the INFILE statement first.
- Specify the same fileref or physical filename in the INFILE and FILE statements.
- Use options that are common to both the FILE and INFILE statements in the INFILE statement. (Any such options that are used in the FILE statement are ignored.)
- Use the SHAREBUFFERS option in the INFILE statement to allow the FILE and INFILE statements to use the same buffer, which saves CPU time and enables you to update individual fields instead of entire records.

Accessing the Contents of the Output Buffer In addition to the _FILE_= variable, you can use the automatic _FILE_ variable to reference the contents of the current output

buffer for the most recent execution of the FILE statement. This character variable is automatically retained and initialized to blanks. Like other automatic variable, _FILE_ is not written to the data set.

When you specify the _FILE_= option in a FILE statement then this variable is also indirectly referenced by the automatic _FILE_ variable. If the automatic _FILE_ variable is present and you omit _FILE_= in a particular FILE statement, then SAS creates an internal _FILE_= variable for that FILE statement. Otherwise, SAS does not create the _FILE_= variable for a particular FILE.

During execution and at the point of reference, the maximum length of this character variable is the maximum length of the current _FILE_= variable. However, because _FILE_ merely references other variables whose lengths are not known until prior to the execution phase, the designated length is 32,767 during the compilation phase. For example, if you assign _FILE_ to a new variable whose length is undefined, the default length of the new variable is 32,767. You can not use the LENGTH statement and the ATTRIB statement to set or override the length of _FILE_. You can use the FORMAT statement and the ATTRIB statement to assign a format to _FILE_.

Like other SAS variables, you can update the _FILE_ variable. The two methods are

FILE = < *'quoted string'| character expression*>

This assignment statement updates the contents of the current output buffer and sets the buffer length to the length of *'quoted string'* or *character expression*. However, this does not affect the PUT statement's current column pointer. The next PUT statement for this FILE statement will begin to update the buffer at column 1 or at the last known location when trailing @ is used in the PUT statement. For example,

```
file print;
_file_='_FILE_';
put 'This is PUT';
```

outputs **`This is PUT`** while

```
file print;
_file_='This is from FILE, oh yeah';
put @14 'both';
```

outputs **`This is from both, oh yeah`**.

a PUT statement

The PUT statement will update the _FILE_ variable because the PUT statement formats data in the output buffer and _FILE_ points to that buffer. However, by default SAS clears the output buffers after a PUT statement executes and outputs the current record (or N= block of records). Therefore, if you wish to examine or further modify the contents of _FILE_ before it is output include a trailing @ or @@ in any PUT statement (when N=1). For other values of N=, use a trailing @ or @@ in any PUT statement where the last line pointer location is on the last record of the record block . In the following code when N=1

```
file ABC;
put 'Something' @;
Y=trim(_file_)||' is here';
file ABC;
put 'Nothing' ;
y=trim(_file_)||' is here';
put y;
```

Y is first assigned **`Something is here`** then Y is assigned **`is here`**.

Any modification of the _FILE_ directly modifies the current output buffer for the current FILE statement. The execution of any subsequent PUT statements for this FILE statement will output the contents of the modified buffer.

FILE only accesses the contents of the current output buffer for a FILE statement, even when you use the N= option to specify multiple buffers. You can access all the N= buffers, but you must use a PUT statement with the # line pointer control to make the desired buffer the current output buffer.

Comparisons

- The FILE statement specifies the *output* file for PUT statements. The INFILE statement specifies the *input* file for INPUT statements.
- Both the FILE and INFILE statements allow you to use options that provide SAS with additional information about the external file being used.
- In the Program Editor, Log, and Output windows, the FILE command specifies an external file and writes the contents of the window to the file.

Examples

Example 1: Executing Statements When a New Page Is Begun This DATA step illustrates how to use the HEADER= option:

- *Write a report.* Use DATA _NULL_ to write a report rather than create a data set.

```
data _null_;
   set sprint;
   by dept;
```

- *Route output to the standard print file. Point to the header information.* The PRINT fileref routes output to the same location as procedure output. HEADER= points to the label that precedes the statements that create the header for each page:

```
file print header=newpage;
```

- *Start a new page for each department:*

```
if first.dept then put _page_;
   put @22 salesrep @34 salesamt;
```

- *Write a header on each page.* These statements execute each time a new page is begun. RETURN is necessary before the label and as the final statement in a labeled group:

```
return;
   newpage:
      put @20 'Sales for 1989' /
          @20 dept=;
      return;
run;
```

Example 2: Determining New Page by Lines Left on the Current Page This DATA step demonstrates using the LINESLEFT= option to determine where the page break should occur, according to the number of lines left on the current page.

- *Write a report.* Use DATA _NULL_ to write a report rather than create a data set:

```
data _null_;
   set info;
```

- *Route output to the standard print file.* The PRINT fileref routes output to the same location as procedure output. LINESLEFT indicates that the variable REMAIN contains the number of lines left on the current page:

```
file print linesleft=remain pagesize=20;
   put @5 name @30 phone
       @35 bldg @37 room;
```

- *Begin a new page when there are fewer than 7 lines left on the current page.* Under this condition, PUT _PAGE_ begins a new page and positions the pointer at line 1:

```
if remain<7 then put _page_ ;
run;
```

Example 3: Arranging the Contents of an Entire Page This example shows use of N=PAGESIZE in a DATA step to produce a two-column telephone book listing, each column containing a name and a phone number:

- *Create a report and write it to a print file.* Use DATA _NULL_ to write a report rather than create a data set. PRINT writes carriage control characters to the output file. N=PAGESIZE makes the entire page available to the output pointer:

```
data _null_;
   file 'external-file' print n=pagesize;
```

- *Specify the columns for the report.* This DO loop iterates twice on each DATA step iteration. The COL value is 1 on the first iteration and 40 on the second:

```
do col=1, 40;
```

- *Write 20 lines of data.* This DO loop iterates 20 times to write 20 lines in column 1. When finished, the outer loop sets COL equal to 40, and this DO loop iterates 20 times again, writing 20 lines of data in the second column. The values of LINE and COL, which are set and incremented by the DO statements, control where the PUT statement writes the values of NAME and PHONE on the page:

```
do line=1 to 20;
   set info;
   put #line @col name $20. +1 phone 4.;
end;
```

- *After composing 2 columns of data, write the page.* This END statement ends the outer DO loop. The PUT _PAGE_ writes the current page and moves the pointer to the top of a new page:

```
end;
   put _page_;
run;
```

Example 4: Identifying the Current Output File This DATA step causes a file identification message to print in the log and assigns the value of the current output file to the variable MYOUT. The PUT statement, demonstrating the assignment of the proper value to MYOUT, writes the value of that variable to the output file:

```
data _null_;
   length myout $ 200;
   file file-specification filename=myout;
   put myout=;
   stop;
run;
```

The PUT statement writes a line to the current output file that contains the physical name of the file:

```
MYOUT=your-output-file
```

Example 5: Dynamically Changing the Current Output File This DATA step uses the FILEVAR= option to dynamically change the currently opened output file to a new physical file.

- *Write a report. Create a long character variable.* Use DATA _NULL_ to write a report rather than create a data set. The LENGTH statement creates a variable with length long enough to contain the name of an external file:

```
data _null_;
   length name $ 200;
```

- *Read an in-stream data line and assign a value to the NAME variable:*

```
   input name $;
```

- *Close the current output file and open a new one when the NAME variable changes.* The *file-specification* is just a place holder; it can be any valid SAS name:

```
   file file-specification filevar=name mod;
      date = date();
```

- *Append a log record to currently open output file:*

```
   put 'records updated ' date date.;
```

- *Supply the names of the external files:*

```
datalines;
external-file-1
external-file-2
external-file-3
;
```

Example 6: When the Output Line Exceeds the Line Length of the Output File Because the combined lengths of the variables are longer than the output line (80 characters), this PUT statement automatically writes three separate records:

```
file file-specification linesize=80;
   put name $ 1-50 city $ 71-90 state $ 91-104;
```

The value of NAME appears in the first record, CITY begins in the first column of the second record, and STATE in the first column of the third record.

Example 7: Reading Data and Writing Text Through a TCP/IP Socket This example shows reading raw data from a file through a TCP/IP socket. The NBYTE= option is used in the INFILE statement:

```
/* Start this first as the server */

   filename serve socket ':5205' server
            recfm=s
            lrecl=25 blocksize=2500;

data _null_;
   nb=25;
   infile serve nbyte=nb;
```

```
   input text $char25.;
   put _all_;
run;
```

This example shows writing text to a file through a TCP/IP socket:

```
/* While the server test is running,*/
/*continue with this as the client. */

filename client socket "&hstname:5205"
         recfm=s
         lrecl=25 blocksize=2500;

data _null_;
   file client;
   put 'Some text to length 25...';
run;
```

See Also

Statements:

“FILE, ODS” on page 815
“FILENAME” on page 821
“INFILE” on page 857
“LABEL” on page 906
“PUT” on page 962
“RETURN” on page 1004
“TITLE” on page 1020

FILE, ODS

Defines the structure of the data component that holds the results of the DATA step and binds that component to a template to produce an output object. ODS sends this object to all open ODS destinations, each of which formats the object appropriately. Also controls what happens when the PUT statement tries to write past the end of a line.

Valid: in a DATA step

Category: File-handling

Type: Executable

Requirement: If you use the ODS option, you must use the fileref PRINT in the FILE statement.

Restriction: The DELIMITER= and DSD options have no effect on the ODS option. You cannot use _FILE_=, FILEVAR=, HEADER=, or PAD with the ODS option.

Syntax

FILE PRINT <ODS<=(*ODS-option(s)*)>><*overflow-control*><N=1 | *pagesize*>;

Arguments

PRINT
is a reserved fileref that directs the output that is produced by any PUT statements to procedure output.

Restriction: You must use PRINT in a FILE statement that uses the ODS option.

Options

N=1 | PAGESIZE
specifies the number of lines that are available to the output pointer in the current iteration of the DATA step.

1
specifies that the buffer can hold only one line at a time. Any code that moves the pointer to a new line causes SAS to write the contents of the buffer to the data component. Once the data are in the data component, you cannot alter them with the PUT statement.

PAGESIZE
specifies that the buffer can hold the whole page at once.
Alias: PS

ODS<=(*ODS-suboptions*)>
defines the structure of the data component that holds the results of the DATA step and binds that component to a template to produce an output object. ODS sends this object to all open ODS destinations, each of which formats the object appropriately. For information on the ODS-suboptions, see "ODS Suboptions" on page 817.

Default: If you do not specify any suboptions, the DATA step uses a default template (base.datastep.table) that is stored in the SASHELP data library. This template defines two generic columns: one for character variables, and one for numeric variables. ODS associates each variable in the DATA step with one of these columns and displays the variables in the order in which they are defined in the DATA step.

Without suboptions, the default template uses the variable's label as its column header. If no label exists, the template uses the variable's name as the column header.

Restriction: You must use the fileref PRINT with the ODS option.

overflow-control
determines what the PUT statement does when pointer controls tell it to write past the last column in the buffer. *Overflow-control* is one of the following:

DROPOVER
does not write items that are directed to columns beyond the last column in the buffer. A message in the log at the end of the DATA step informs you if data were not written to the buffer.

FLOWOVER
moves the pointer to a new line when an item is directed to a column beyond the last column in the buffer, writes the next item in the first column of the new line, and continues to process the PUT statement.

STOPOVER
stops processing the DATA step immediately if a PUT statement attempts to write to a column beyond the last column in the buffer. SAS discards the

data item, writes the portion of the line that was built before the error occurred, and issues an error message.

Default: FLOWOVER

ODS Suboptions

You can use any combination of the following suboptions with the ODS= option in the FILE statement:

To do this...	Use this suboption
Specify one or more columns for the data component	COLUMNS= or VARIABLES=
Specify default values for column attributes that exist in the template but that get their values from the data component	DYNAMIC=
Assert that all column definitions in the template can or cannot be used by more than one variable	GENERIC=
Specify a column header to use for any column that does not have a column header specified in the COLUMNS= or VARIABLES= suboption	LABEL=
Specify a name for the output object that the DATA step produces	OBJECT=
Specify a label for the output object that the DATA step produces	OBJECTLABEL=
Specify the template to use with the data component to produce the output object	TEMPLATE=

COLUMNS=(*column-specification(s)*)
specifies one or more columns for the data component. Each *column-specification* associates a DATA step variable with a column that is defined in the template. The order of the columns in the data component is determined by their order in the COLUMNS= suboption.

Note: By default, the order of the columns in the output object is determined by their order in the template, not by their order in the data component. You can override this order by using the ORDER_DATA option in the PROC TEMPLATE step that creates the template. The default DATA step template uses this option. △

Each *column-specification* has this general form:
column-name<=*variable-name*><(*attribute(s)*)>

column-name
is the name of a column. This name must match a name that is defined in the template that you use.

Restriction: *column-name* must conform to the rules for SAS variable names. The first character must be a letter or an underscore. Subsequent characters can be letters, numbers, or underscores.

Tip: You can use list notation (for example, **score1-score5**) to specify a range of column names.

variable-name
specifies a variable in the DATA step to place in the specified column.

Default: If you omit *variable-name*, ODS looks for a DATA step variable named *column-name* to place in the specified column. If no such variable exists, ODS returns an error.

Tip: You can use list notation (for example, **score1-score5**) to specify a range of variable names.

attribute
assigns a characteristic, such as a label or a format, to a particular column in the data component. These individual specifications override any attributes that are set, in the DATA step, for the entire data component. You assign attributes with the following suboptions.

DYNAMIC=*dynamic-specification(s)*
specifies a value for a column attribute that exists in the template but that gets its value from the data component. For details, see the discussion of DYNAMIC= on page 818.

FORMAT=*format-name*
specifies a format for this particular column.

Default: ODS uses the first of these formats for the variable that it finds:

- a format that is specified in a column definition in a template.
- a format that is specified in the FORMAT= column attribute
- a format that is specified in a FORMAT statement
- the default format ($*w*. for character variables; *BEST12*. for numeric variables)

GENERIC=ON|OFF
asserts that the DATA step does or does not use this column definition for multiple variables. For details, see GENERIC= on page 819.

LABEL='*column-label*'
specifies a label for this particular column. For details, see LABEL= on page 819.

Default: If you do not specify COLUMNS= or VARIABLES=, the order of columns in the data component matches the order of the corresponding variables in the program data vector.

Restriction: You can use only one COLUMNS= suboption in a FILE statement.

Interaction: You can use either the COLUMNS= suboption or the VARIABLES= suboption to associate variables with columns. However, you cannot use both suboptions in the same FILE statement.

DYNAMIC=(*dynamic-specification(s)*)
specifies default values for dynamic attributes values. Columns that do not contain their own DYNAMIC= specifications use these.

A dynamic attribute value is defined in the template. Its name serves as a placeholder for the value that is supplied to the data component with the DYNAMIC= suboption. When ODS creates the output object from the template and the data component, it substitutes the appropriate value from the data component for the value's name in the template.

Each *dynamic-specification* has the following form:
dynamic-value-name<=*variable-name* | *constant*>

dynamic-value-name
is the name that the template gives to a dynamic attribute value.

variable-name
specifies a variable whose value is assigned to *dynamic-value-name* and passed to ODS to substitute for the placeholder in the template when it creates the output object.

constant
specifies a constant to assign to *dynamic-value-name* and to pass to ODS to substitute for the placeholder in the template when it creates the output object.

Default: If you omit *variable-name* and *constant*, then the DATA step looks for a variable name that matches '*dynamic-value-name*'.

Tip: By default, DYNAMIC= applies to all columns in the data component. You can override this specification for an individual column by specifying DYNAMIC= as an attribute for that column in the COLUMNS= or the VARIABLES= suboption.

GENERIC=ON | OFF
asserts that the DATA step does or does not use all column definitions for multiple variables.

ON
asserts that the DATA step uses all column definitions for multiple variables.

OFF
asserts that the DATA step uses no column definitions for multiple variables.

Default: OFF

Interaction: If you do not specify a template (see TEMPLATE= on page 820), GENERIC= is set to ON.

Restriction: To have ODS recognize the column names as a match, the GENERIC= specification must match the GENERIC= specification in the template that you are using.

Tip: By default, GENERIC= applies to all columns in the data component. You can override this specification for an individual column by specifying GENERIC= as an attribute for that column in the COLUMNS= or the VARIABLES= suboption.

LABEL='*column-label*'
specifies a label for any column that does not have a label specified in the COLUMNS= or VARIABLES= suboption.

Default: ODS uses for the column header the first of these labels that it finds:

- a label that is specified with HEADER= for a particular column in the template.
- a label that is specified for a particular column with LABEL= in the COLUMNS= or VARIABLES= suboption.
- a label that is specified with LABEL= in the ODS= option.
- a label that is assigned with the LABEL statement in the DATA step.
- If no label is specified, the contents of the template determines whether the column header contains the variable name or is blank.

OBJECT=*object-name*
specifies a name for the output object.

The Results window and the HTML contents file both contain a description of and a link to each output object. The description contains the first of these items that ODS finds:

- the object's label (see OBJECTLABEL= on page 820)
- the current title if it is not the default title, "The SAS System"
- the object's name
- the string **`FilePrint#`**, where # increases by 1 for each DATA step that you run in the current SAS process without specifying an object name or an object label.

Restriction: *Object-name* must conform to the rules for SAS variable names. For information on these rules, see "Appendix 1: Rules for Words and Names in the SAS Language" in *SAS Language Reference: Dictionary*.

OBJECTLABEL='*object-label*'
specifies a label for the output object.

The Results window and the HTML contents file both contain a description of and a link to each output object. The description contains the first of these items that ODS finds:

- the object's label
- the current title if it is not the default title, "The SAS System"
- the object's name (see the discussion of OBJECT= on page 819).
- the string **FilePrint#**, where # increases by 1 for each DATA step that you run in the current SAS process without specifying an object name or an object label.

TEMPLATE='*template-name*'
specifies the template to use with the data component to produce the output object.

template-name
is the path to the template. SAS stores a template as an item in an item store. By default, ODS first looks for *template-name* in SASUSER.TEMPLAT. If it doesn't find the template there, it looks in SASHELP.TMPLMST. You can change the locations that it searches with the ODS PATH statement.

Default: The default template (base.datastep.table) that is stored in the SASUSER SAS data library.

Interaction: When you use the default template, GENERIC= is set to on for all columns in the data component. (See GENERIC= on page 819).

VARIABLES=(*variable-specification(s)*)
specifies one or more columns for the data component of the output object. Each *variable-specification* associates a DATA step variable with a column that is defined in the template.

Note: By default, the order of the columns in the output object is determined by their order in the template, not by their order in the data component. You can override this order by using the ORDER_DATA option in the PROC TEMPLATE step that creates the template. The default DATA step template uses this option. △

Each *variable-specification* has this general form:
variable-name<=*column-name*><(*attribute(s)*)>

variable-name
specifies a variable in the DATA step to place in the specified column.

Tip: You can use list notation (for example, **score1-score5**) to specify a range of variable names.

column-name
is the name of a column. This name must match a name that is defined in the template.

Default: If you omit *column-name*, ODS looks for a column in the template that is named *variable-name* and places the variable in that column. If no such column exists, ODS returns an error.

Restriction: *Column-name* must match a column name in the template that you are using. It must also conform to the rules for SAS variable names. For information on these rules, see "Appendix 1: Rules for Words and Names in the SAS Language" in *SAS Language Reference: Dictionary*.

Tip: You can use list notation (for example, **score1-score5**) to specify a range of column names.

attribute
assigns a characteristic, such as a label or a format, to a particular column in the data component. These individual specifications override any attributes that are set, in the DATA step, for the entire data component. You assign attributes just as you do in the COLUMNS= suboption (see the discussion of attributes on page 818).

Restriction: You can use only one VARIABLES= suboption in a FILE statement.

Interaction: You can use either the COLUMNS= suboption or the VARIABLES= suboption to associate variables with columns. However, you cannot use both suboptions in the same FILE statement.

Tip: VARIABLES= is primarily for use with the default DATA step template. When you are using the default template, the DATA step can map variables to the appropriate column in the template so you don't need to specify a column name.

Details

Within the DATA step, the ODS option in the FILE statement and the _ODS_ option in the PUT statement provide connections with the Output Delivery System (ODS). You use both of these connections to route the results of a DATA step to ODS. By default, when the DATA step uses ODS, ODS writes output objects to the procedure output and places links to them in the Results folder. You can use global ODS statements to write to other ODS destinations.

The FILE and PUT statements interact in the following way:

- The ODS option in the FILE statement defines the structure of the data component that holds the results of the DATA step.
- The _ODS_ option in the PUT statement writes values (as specified by the ODS option in the FILE statement) into a special buffer. This buffer is written to the data component.
- The ODS option in the FILE statement binds the data component to a template to produce an output object. ODS sends this object to all open ODS destinations, each of which formats the object appropriately.

The ODS destinations are controlled by the global ODS statements. You can use an existing template or create your own with the TEMPLATE. procedure.

For details on using the Output Delivery System, see *The Complete Guide to the SAS Output Delivery System*.

See Also

Statements:

"FILE" on page 802
"PUT, _ODS_" on page 989

FILENAME

Associates a SAS fileref with an external file or an output device; disassociates a fileref and external file; lists attributes of external files

Valid: anywhere
Category: Data Access

Syntax

❶ **FILENAME** *fileref* <*device-type*> '*external-file*' <*host-options*>;

❷ **FILENAME** *fileref* <*device-type*> <*host-options*>;

❸ **FILENAME** *fileref* CLEAR | _ALL_ CLEAR;

❹ **FILENAME** *fileref* LIST | _ALL_ LIST;

Arguments

fileref
is any SAS name when you are assigning a new fileref. When you are disassociating a currently-assigned fileref or when you are listing file attributes with the FILENAME statement, specify a fileref that was previously assigned with a FILENAME statement or a host-level command.

Tip: The association between a fileref and an external file lasts only for the duration of the SAS session or until you change it or discontinue it with another FILENAME statement. Change the fileref for a file as often as you want.

'*external-file*'
is the physical name of an external file. The physical name is the name that is recognized by the operating environment.

Operating Environment Information: For details on specifying the physical names of external files, see the SAS documentation for your operating environment. △

Tip: Specify *external-file* when you assign a fileref to an external file.

Tip: You can associate a fileref with a single file or with an aggregate file storage location.

device-type
specifies the type of device or the access method that is used if the fileref points to an input or output device or location that is not a physical file:

DISK	specifies that the device is a disk drive. Tip: When you assign a fileref to a file on disk, you are not required to specify DISK. Alias: BASE
DUMMY	specifies that the output to the file is discarded. Tip: Specifying DUMMY can be useful for testing.
GTERM	indicates that the output device-type is a graphics device that will be receiving graphics data.
PIPE	specifies an unnamed pipe. *Note:* Some operating environments do not support pipes. △
PLOTTER	specifies an unbuffered graphics output device.
PRINTER	specifies a printer or printer spool file.

TAPE
: specifies a tape drive.

TEMP
: creates a temporary file that exists only as long as the filename is assigned. The temporary file can be accessed only through the logical name and is available only while the logical name exists.

Restriction: Do not specify a physical pathname. If you do, SAS returns an error.

Tip: Files manipulated by the TEMP device can have the same attributes and behave identically to DISK files.

TERMINAL
: specifies the user's terminal.

Operating Environment Information: Additional specifications may be required when you specify some devices. See the SAS documentation for your operating environment before specifying a value other than DISK. Values in addition to the ones listed here may be available in some operating environments. △

CLEAR
: disassociates one or more currently assigned filerefs.

Tip: Specify *fileref* to disassociate a single fileref. Specify _ALL_ to disassociate all currently assigned filerefs.

ALL
: specifies that the CLEAR or LIST argument applies to all currently-assigned filerefs.

LIST
: writes the attributes of one or more files to the SAS log.

Interaction: Specify *fileref* to list the attributes of a single file. Specify _ALL_ to list the attributes of all files that have filerefs in your current session.

Host Options

Host-options specify details, such as file attributes and processing attributes, that are specific to your operating environment.

Operating Environment Information: For a list of valid specifications, see the SAS documentation for your operating environment. △

Details

Host Information

Operating Environment Information: Using the FILENAME statement requires host-specific information. See the SAS documentation for your operating environment before using this statement. Note also that host commands are available in some operating environments that associate a fileref with a file and that break that association. △

Definitions

external file
: is a file that is created and maintained in the operating environment from which you need to read data, SAS programming statements, or autocall macros or to which you want to write output. An external file can be a single file or an aggregate storage location that contains many individual external files.

For an example, see Example 3 on page 825.

Operating Environment Information: Different operating environments call an aggregate grouping of files by different names, such as a directory, a MACLIB, or a partitioned data set. For details on specifying external files, see the SAS documentation for your operating environment. △

fileref
(a file reference name) is a shorthand reference to an external file. Once you have associated a fileref with an external file, you can use it as a shorthand reference for that file in SAS programming statements (such as INFILE, FILE, and %INCLUDE) and in other commands and statements in SAS software that access external files.

❶Associating a Fileref with an External File Use this form of the FILENAME statement to associate a fileref with an external file on disk:

FILENAME *fileref 'external-file'* <*host-options*>;

To associate a fileref to a file other than a disk file, you may need to specify a device type, depending on your operating environment, as shown in this form:

FILENAME *fileref* <*device-type*> <*host-options*>;

The association between a fileref and an external file lasts only for the duration of the SAS session or until you change it or discontinue it with another FILENAME statement. Change the fileref for a file as often as you want.

❷Associating a Fileref with a Terminal, Printer, or Plotter To associate a fileref with an output device, use this form:

FILENAME *fileref device-type* <*host-options*>;

❸Disassociating a Fileref from an External File To disassociate a fileref from a file, use a FILENAME statement, specifying the fileref and the CLEAR option.

❹Writing File Attributes to the SAS Log Use a FILENAME statement to write the attributes of one or more external files to the SAS log. Specify *fileref* to list the attributes of one file; use _ALL_ to list the attributes of all files that have been assigned filerefs in your current SAS session.

FILENAME *fileref* LIST | _ALL_ LIST;

Comparisons

The FILENAME statement assigns a fileref to an external file. The LIBNAME statement assigns a libref to a SAS data set or to a DBMS file that can be accessed like a SAS data set.

Examples

Example 1: Specifying a Fileref or a Physical File Name You can specify an external file either by associating a fileref with the file and then specifying the fileref or by specifying the physical file name in quotation marks:

```
filename sales 'your-input-file';

data jansales;
      /* specifying a fileref */
```

```
   infile sales;
   input salesrep $20. +6 jansales febsales
         marsales;
run;

data jansales;
      /* physical filename in quotes */
   infile 'your-input-file';
   input salesrep $20. +6 jansales febsales
         marsales;
run;
```

Example 2: Using a FILENAME and a LIBNAME Statement

This example reads data from a file that has been associated with the fileref GREEN and creates a permanent SAS data set stored in a SAS data library that has been associated with the libref SAVE.

```
filename green 'your-input-file';
libname save 'SAS-data-library';

data save.vegetabl;
   infile green;
   input lettuce cabbage broccoli;
run;
```

Example 3: Associating a Fileref with an Aggregate Storage Location If you associate a fileref with an aggregate storage location, use the fileref, followed in parentheses by an individual filename, to read from or write to any of the individual external files stored there.

Operating Environment Information: Some operating environments allow you to read from but not write to members of aggregate storage locations. See the SAS documentation for your operating environment. △

In this example each DATA step reads from an external file (REGION1and REGION2, respectively) that is stored in the same aggregate storage location and that is referenced by the fileref SALES.

```
filename sales 'aggregate-storage-location';

data total1;
   infile sales(region1);
   input machine $ jansales febsales marsales;
   totsale=jansales+febsales+marsales;
run;

data total2;
   infile sales(region2);
   input machine $ jansales febsales marsales;
   totsale=jansales+febsales+marsales;
run;
```

Example 4: Routing PUT Statement Output In this example, the FILENAME statement associates the fileref OUT with a printer that is specified with a host-dependent option, and the FILE statement directs PUT statement output to that printer.

```
filename out printer host-options;

data sales;
   file out print;
   input salesrep $20. +6 jansales
         febsales marsales;
   put _infile_;
   datalines;
Jones, E. A.              124357 155321 167895
Lee, C. R.                111245 127564 143255
Desmond, R. T.             97631 101345 117865
;
```

You can use the FILENAME and FILE statements to route PUT statement output to several different devices during the same session. To route PUT statement output to your display monitor, use the TERMINAL option in the FILENAME statement, as shown here:

```
filename show terminal;

data sales;
   file show;
   input salesrep $20. +6 jansales
         febsales marsales;
   put _infile_;
   datalines;
Jones, E. A.              124357 155321 167895
Lee, C. R.                111245 127564 143255
Desmond, R. T.             97631 101345 117865
;
```

See Also

Statements:

“FILE” on page 802
“%INCLUDE” on page 851
“INFILE” on page 857
“FILENAME, CATALOG Access Method” on page 826
“FILENAME, FTP Access Method” on page 829
“FILENAME, SOCKET Access Method” on page 835
“LIBNAME” on page 913

SAS Windowing Interface Commands:

FILE and INCLUDE

FILENAME, CATALOG Access Method

References a SAS catalog as an external file

Valid: anywhere

Category: Data Access

Syntax

FILENAME *fileref* **CATALOG** *'catalog'* <*catalog-options*>;

Arguments

fileref
is a valid fileref.

CATALOG
specifies the access method that enables you to reference a SAS catalog as an external file. You can then use any SAS commands, statements, or procedures that can access external files to access a SAS catalog.

Tip: This access method makes it possible for you to invoke an autocall macro directly from a SAS catalog.

Tip: With this access method you can read any type of catalog entry, but you can write only to entries of type LOG, OUTPUT, SOURCE, and CATAMS.

Tip: If you want to access an entire catalog (instead of a single entry), you must specify its two-level name in the *catalog* parameter.

Alias: LIBRARY

'catalog'
is a valid two-, three-, or four-part SAS catalog name, where the parts represent *library.catalog.entry.entrytype*.

Default: The default entry type is CATAMS, with one exception. When you reference that fileref with %INCLUDE, the type is assumed to be SOURCE.

Restriction: The CATAMS entry type is used only by the CATALOG access method. The CPORT and CIMPORT procedures do not support this entry type.

Catalog Options

Catalog-options can be any of the following:

LRECL=*lrecl*
where *lrecl* is the maximum record length for the data in bytes.

Default: For input, the actual LRECL value of the file is the default. For output, the default is 132.

RECFM=*recfm*
where *recfm* is the record format.

Range: V (variable), F (fixed), and P (print).

Default: V

DESC=*description*
where *description* is a text description of the catalog.

MOD
specifies to append to the file.

Default: If you omit MOD, the file is replaced.

Details

The CATALOG access method in the FILENAME statement enables you to reference a SAS catalog as an external file. You can then use any SAS commands, statements, or procedures that can access external files to access a SAS catalog. As an example, the catalog access method makes it possible for you to invoke an autocall macro directly from a SAS catalog. See Example 5 on page 829.

With the CATALOG access method you can read any type of catalog entry, but you can only write to entries of type LOG, OUTPUT, SOURCE, and CATAMS. If you want to access an entire catalog (instead of a single entry), you must specify its two-level name in the *catalog* argument.

Examples

Example 1: Using %INCLUDE with a Catalog Entry This example submits the source program that is contained in SASUSER.PROFILE.SASINP.SOURCE:

```
filename fileref1
         catalog 'sasuser.profile.sasinp.source';
%include fileref1;
```

Example 2: Using %INCLUDE with Several Entries in a Single Catalog This example submits the source code from three entries in the catalog MYLIB.INCLUDE. When no entry type is specified, the default is CATAMS.

```
filename dir catalog 'mylib.include';
%include dir(mem1);
%include dir(mem2);
%include dir(mem3);
```

Example 3: Reading and Writing a CATAMS Entry This example uses a DATA step to write data to a CATAMS entry, and another DATA step to read it back in:

```
filename mydata
         catalog 'sasuser.data.update.catams';

   /* write data to catalog entry update.catams */
data _null_;
   file mydata;
   do i=1 to 10;
      put i;
   end;
run;

   /* read data from catalog entry update.catams */
data _null_;
   infile mydata;
   input;
   put _INFILE_;
run;
```

Example 4: Writing to a SOURCE Entry This example writes code to a catalog SOURCE entry and then submits it for processing:

```
filename incit
         catalog 'sasuser.profile.sasinp.source';
```

```
data _null_;
   file incit;
   put 'proc options; run;';
run;

%include incit;
```

Example 5: Executing an Autocall Macro from a SAS Catalog If you store an autocall macro in a SOURCE entry in a SAS catalog, you can point to that entry and invoke the macro in a SAS job. Use these steps:

1. Store the source code for the macro in a SOURCE entry in a SAS catalog. The name of the entry is the macro name.
2. Use a LIBNAME statement to assign a libref to that SAS library.
3. Use a FILENAME statement with the CATALOG specification to assign a fileref to the catalog: *libref.catalog*.
4. Use the SASAUTOS= option and specify the fileref so that the system knows where to locate the macro. Also set MAUTOSOURCE to activate the autocall facility.
5. Invoke the macro as usual: *%macro-name*.

This example points to a SAS catalog named MYSAS.MYCAT. It then invokes a macro named REPORTS, which is stored as a SAS catalog entry named MYSAS.MYCAT.REPORTS.SOURCE:

```
libname mysas 'SAS-data-library';
filename mymacros catalog 'mysas.mycat';
options sasautos=mymacros mautosource;

%reports
```

See Also

Statements:

"FILENAME" on page 821
"FILENAME, FTP Access Method" on page 829
"FILENAME, SOCKET Access Method" on page 835

FILENAME, FTP Access Method

Allows you to access remote files using the FTP protocol

Valid: anywhere
Category: Data Access

Syntax

FILENAME *fileref* FTP *'external-file'* <*ftp-options*>;

Arguments

fileref
is a valid fileref.

Tip: The association between a fileref and an external file lasts only for the duration of the SAS session or until you change it or discontinue it with another FILENAME statement. You can change the fileref for a file as often as you want.

FTP
specifies the access method that enables you to use file transfer protocol (FTP) to read or write to a file from any host machine that you can connect to on a network with an FTP server running.

Tip: Use FILENAME with FTP when you want to connect to the host machine, to log in to the FTP server, to make records in the specified file available for reading or writing, and then to disconnect from the host machine.

'external-file'
specifies the physical name of an external file that you want to read from or write to. The physical name is the name that is recognized by the operating environment.

Operating Environment Information: For details on specifying the physical names of external files, see the SAS documentation for your operating environment. △

Tip: If you are not transferring a file but performing a task such as retrieving a directory listing, you do not need to specify a filename. Instead, put empty quotation marks in the statement. See Example 1 on page 833.

Tip: You can associate a fileref with a single file or with an aggregate file storage location.

host-options
specify host-specific details such as file attributes and processing attributes.

Operating Environment Information: For a list of valid specifications, see the SAS documentation for your operating environment. △

FTP-Options

BLOCKSIZE=*blocksize*
where *blocksize* is the size of the data buffer in bytes.

Default: 8192

CD='*directory*'
issues a command that changes the working directory for the file transfer to the *directory* specified.

DEBUG
writes to the SAS log informational messages that are sent to and received from the FTP server.

HOST='*host*'
where *host* is the network name of the remote host with the FTP server running.

Range: You can specify either the name of the host (for example, **`server.pc.sas.com`**) or the IP address of the machine (for example, **`190.96.6.96`**).

LIST
issues the LIST command to the FTP server. LIST returns the contents of the working directory as records that contain all of the file attributes that are listed for each file.

Tip: The file attributes that are returned will vary, depending on the FTP server that is being accessed.

LRECL=*lrecl*
where *lrecl* is the logical record length.

Default: 256

LS
issues the LS command to the FTP server. LS returns the contents of the working directory as records with no file attributes.

Tip: The file attributes that are returned will vary, depending on the FTP server that is being accessed.

Tip: To return a listing of a subset of files, use the LSTFILE= option in addition to LS.

LSFILE='*character-string*'
in combination with the LS option, specifies a character string that allows you to request a listing of a subset of files from the working directory. Enclose the character string in quotation marks.

Restriction: LSFILE= can be used only if LS is specified.

Tip: You can specify a wildcard as part of '*character-string* '.

Tip: The file attributes that are returned will vary, depending on the FTP server that is being accessed.

Example: This statement lists all files that start with **sales** and end with **sas**:

```
filename foo ftp '' ls lsfile='sales*.sas'
        other-ftp-options;
```

MGET
transfers multiple files, similar to the FTP command MGET.

Tip: The whole transfer is treated like one file. However, as the transfer of each new file is started, the EOV= variable is set to 1.

Tip: Specify MPROMPT to prompt the user before each file is sent.

MPROMPT
specifies whether to prompt for confirmation that a file is to be read, if necessary, when the user executes the MGET option.

PASS='*password*'
where *password* is the password to use with the user name specified in the USER= option.

Tip: You can specify the PROMPT option instead of the PASS option, which tells the system to prompt you for the password.

Tip: If the user name is **anonymous**, the remote host may require that you specify your e-mail address as the password.

PROMPT
specifies to prompt for the user login password, if necessary.

Interaction: If PROMPT is specified without USER=, then the user is prompted for an id, as well as a password.

RCMD= '*command* '
where *command* is the command to send to the FTP server.

Restriction: For an IBM FTP server, this argument with the following command is required:

```
rcmd='site rdw'
```

The SITE RDW command is required for an IBM FTP server because, by default, this server removes the RDW (record descriptor word) that exists in the record before it is transferred. Issuing the SITE RDW command causes the IBM FTP server to include the RDW as part of the data.

RECFM=*recfm*
where *recfm* is one of three record formats:

F is fixed record format. Thus, all records are of size LRECL with no line delimiters. Data are transferred in image (binary) mode.

S is stream record format. Data are transferred in image (binary) mode.

Interaction: The amount of data read is controlled by the current LRECL value or by the value of the NBYTE= variable in the FILE or INFILE statement. The NBYTE= option specifies a variable that is equal to the amount of data to be read. This amount must be less than or equal to LRECL.

See Also: The NBYTE= option on page 808 in the FILE statement and the NBYTE= option on page 862in the INFILE statement.

V is variable record format (the default). In this format, records have varying lengths, and they are transferred in text (stream) mode.

Interaction: Any record larger than LRECL is truncated.

Tip: If you are using files with the IBM 370 Variable format or the IBM 370 Spanned Variable format and using an IBM Pascal FTP server, use the S370V or S370VS options instead of the RECFM= option. See S370V and S370VS below.

Default: V

RHELP
issues the HELP command to the FTP server. The results of this command are returned as records.

RSTAT
issues the RSTAT command to the FTP server. The results of this command are returned as records.

S370V
indicates that the file being read is in IBM 370 variable format.

Restriction: This option works only with an IBM Pascal FTP server.

Interaction: If you specify this option and the RECFM= option, SAS ignores the RECFM= option.

Tip: The data are transferred in image or binary format, and are in local data format. Thus, you must use appropriate SAS informats to read the data correctly on non-EBCDIC hosts.

S370VS
indicates that the file that is being read is in IBM 370 variable-spanned format.

Restriction: This option works only with an IBM Pascal FTP server.

Interaction: If you specify this option and the RECFM= option, SAS ignores the RECFM= option.

Tip: The data are transferred in image or binary format and are in local data format. Thus, you must use appropriate SAS informats to read the data correctly on non-EBCDIC hosts.

USER='*username*'
where *username* is used to log in to the FTP server.

Interaction: If PROMPT is specified, but USER= is not, the user is prompted for an id.

Comparisons

As with the FTP **get** and **put** commands, the FTP access method lets you download and upload files; however, this method directly reads files into your SAS session without first storing them on your system.

Examples

Example 1: Retrieving a Directory Listing This example retrieves a directory listing from a host named **mvshost1** for user **dilbert**, and prompts **dilbert** for a password:

```
filename dir ftp '' ls user='dilbert'
         host='mvshost1.mvs.sas.com' prompt;

data _null_;
   infile dir;
   input;
   put _INFILE_;
run;
```

Note: The quotation marks are empty because no file is being transferred; because they are required by the syntax, however, the statement includes empty quotation marks. △

Example 2: Reading a File from a Remote Host This example reads a file called **data** in the directory **/u/kudzu/mydata** from the remote UNIX host **hp720**:

```
filename myfile ftp 'data' cd='/u/kudzu/mydata'
         user='guest' host='hp720.hp.sas.com'
         recfm=v prompt;

data mydata / view=mydata;   /* Create a view */
   infile myfile;
   input x $10. y 4.;
run;

proc print data=mydata;     /* Print the data */
run;
```

Example 3: Creating a File on a Remote Host This example creates a file called **test.dat** in a directory called **c:\remote** for the user **bbailey** on the host **winnt.pc**.

```
filename create ftp 'c:\remote\test.dat'
         host='winnt.pc'
         user='bbailey' prompt recfm=v;
```

```
data _null_;
   file create;
   do i=1 to 10;
      put i=;
   end;
run;
```

Example 4: Reading an S370V-Format File on an OS/390 Host This example reads an S370V-format file from an OS/390 system. See the RCMD= on page 831 option for more information on RCMD='**site rdw**'.

```
filename viewdata ftp 'sluggo.stat.data'
         user='sluggo' host='mvshost1'
         s370v prompt rcmd='site rdw';

data mydata / view=mydata;   /* Create a view */
   infile viewdata;
   input x $ebcdic8.;
run;

proc print data=mydata;     /* Print the data */
run;
```

Example 5: Anonymously Logging In to FTP This example shows how to login to FTP anonymously, if the host accepts anonymous logins.

Note: Some anonymous FTP servers require a password. If required, your e-mail address is usually used. See PASS= on page 831 under FTP-Options. △

```
filename anon ftp '' ls host='130.96.6.1'
         user='anonymous';

data _null_;
   infile anon;
   input;
   list;
run;
```

Note: The quotation marks following the argument FTP are empty. A filename is needed only when transferring a file, not when routing a command. The quotation marks, however, are required. △

Example 6: Importing a Transport Dataset

This example uses the CIMPORT procedure to import a transport data set from a host named **mvshost1** for user **calvin**. The new data set will reside locally in the SASUSER library. Note that user and password can be SAS macro variables. If you specify a full-qualified data set name, use double quotation marks and single quotation marks. Otherwise, the system will append the profile prefix to the name that you specify.

```
   %let user=calvin;
   %let pw=xxxxx;
   filename inp ftp "'calvin.mat1.cpo'" user="&user"
            pass="&pw" rcmd='binary'
            host='sdcmvs.mvs.sas.com';

  proc cimport library=sasuser infile=inp;
  run;
```

Example 7: Transporting a SAS Data Library This example uses the CPORT procedure to transport a SAS data library to a host named **mvshost1** for user **calvin**. It will create a new sequential file on the host called **userid.mat64.cpo** with the recfm of **fb**, lrecl of 80, and blocksize of 8000.

```
filename inp ftp 'mat64.cpo' user='calvin'
         pass="xxxx" host='mvhost1'
         lrecl=80 recfm=f blocksize=8000
     rcmd='site blocksize=8000 recfm=fb lrecl=80';

proc cport library=mylib file=inp;
run;
```

Example 8: Creating a Transport Library with Transport Engine This example creates a new SAS data library on host **mvshost1**. The FILENAME statement assigns a fileref to the new data set. Note the use of the RCMD= option to specify important file attributes. The LIBNAME statement uses a libref that is the same as the fileref and assigns it to the XMPORT engine. The PROC COPY step copies all data sets from the SAS data library referenced by MYLIB to the XPORT engine. Output from the PROC CONTENTS step confirms that the copy was successful.

```
filename inp ftp  'mat65.cpo' user='calvin'
         pass="xxxx"  host='mvshost1'
         lrecl=80 recfm=f blocksize=8000
     rcmd='site blocksize=8000 recfm=fb lrecl=80';

libname mylib 'SAS-data-library';
libname inp xport;

proc copy in=mylib out=inp mt=data;
run;

proc contents data=inp._all_;
run;
```

See Also

Statements:

"FILENAME" on page 821
"FILENAME, CATALOG Access Method" on page 826
"FILENAME, SOCKET Access Method" on page 835
"FILENAME, URL Access Method" on page 839
"LIBNAME" on page 913

FILENAME, SOCKET Access Method

Allows you to read from or write to a TCP/IP socket

Valid: anywhere

Category: Data Access

Syntax

❶ **FILENAME** *fileref* SOCKET '*hostname*:*portno*'
<*tcpip-options*>;

❷ **FILENAME** *fileref* SOCKET ':*portno*' SERVER
<*tcpip-options*>;

Arguments

fileref
is a valid fileref.

Tip: The association between a fileref and an external file lasts only for the duration of the SAS session or until you change it or discontinue it with another FILENAME statement. You can change the fileref for a file as often as you want.

SOCKET
specifies the access method that enables you to read from or write to a Transmission Control Protocol/Internet Protocol (TCP/IP) socket.

'*hostname*:*portno*'
is the name or IP address of the host and the TCP/IP port number to connect to.

Tip: Use this specification for client access to the socket.

':*portno*'
is the port number to create for listening.

Tip: Use this specification for server mode.

Tip: If you specify **:0**, the system will choose a number.

SERVER
sets the TCP/IP socket to be a listening socket, thereby enabling the system to act as a server that is waiting for a connection.

Tip: The system accepts all connections serially; only one connection is active at any one time.

See Also: The RECONN= option description on page 837 under *TCPIP-Options*.

TCPIP-Options

BLOCKSIZE=*blocksize*
where *blocksize* is the size of the socket data buffer in bytes.

Default: 8192

LRECL=*lrecl*
where *lrecl* is the logical record length.

Default: 256

RECFM=*recfm*
where *recfm* is one of three record formats:

F	is fixed record format. Thus, all records are of size LRECL with no line delimiters. Data are transferred in image (binary) mode.
S	is stream record format.

Tip: Data are transferred in image (binary) mode.

Interactions: The amount of data that is read is controlled by the current LRECL value or the value of the NBYTE= variable in the FILE or INFILE statement. The NBYTE= option specifies a variable equal to the amount of data to be read. This amount must be less than or equal to LRECL.

See Also: The NBYTE= option on page 808 under the FILE statement and the NBYTE= option on page 862 in the INFILE statement.

V is variable record format (the default).

Tip: In this format, records have varying lengths, and they are transferred in text (stream) mode.

Tip: Any record larger than LRECL is truncated.

Default: V

RECONN=*conn-limit*
where *conn-limit* is the maximum number of connections that the server will accept.

Explanation: Because only one connection may be active at a time, a connection must be disconnected before the server can accept another connection. When a new connection is accepted, the EOV= variable is set to 1. The server will continue to accept connections, one at a time, until *conn-limit* has been reached.

TERMSTR='*eol-char*'
where *eol-char* is the line delimiter to use when RECFM=V. There are three valid values:

CRLF carriage return (CR) followed by line feed (LF).

LF line feed only (the default).

NULL NULL character (0x00).

Default: LF

Restriction: Use this option only when RECFM=V.

Details

The Basics A TCP/IP socket is a communication link between two applications. The *server* application creates the socket and waits for a connection. The *client* application connects to the socket. With the SOCKET access method, you can use SAS to communicate with another application over a socket in either client or server mode. The client and server applications can reside on the same machine or on different machines that are connected by a network.

As an example, you can develop an application using Microsoft Visual Basic that communicates with a SAS session that uses the TCP/IP sockets. Note that Visual Basic does not provide inherent TCP/IP support. You can obtain a custom control (VBX) from SAS Institute Technical Support (free of charge) that allows a Visual Basic application to communicate through the sockets.

❶Using the SOCKET Access Method in Client Mode

In client mode, a local SAS application can use the SOCKET access method to communicate with a remote application that acts as a server (and waits for a connection). Before you can connect to a server, you must know:

- the network name or IP address of the host machine running the server.
- the port number that the remote application is listening to for new connections.

The remote application can be another SAS application, but it doesn't need to be. When the local SAS application connects to the remote application through the TCP/IP socket, the two applications can communicate by reading from and writing to the socket as if it were an external file. If at any time the remote side of the socket is disconnected, the local side will also automatically terminate.

❷Using the SOCKET Access Method in Server Mode When the local SAS application is in server mode, it remains in a wait state until a remote application connects to it. To use the SOCKET access method in server mode, you need to know only the port number that you want the server to listen to for a connection. Typically, servers use *well-known ports* to listen for connections. These port numbers are reserved by the system for specific server applications. For more information about how well-known ports are defined on your system, refer to the documentation for your TCP/IP software or ask your system administrator.

If the server application does not use a well-known port, then the system assigns a port number when it establishes the socket from the local application. However, because any client application that waits to connect to the server must know the port number, you should try to use a well-known port.

While a local SAS server application is waiting for a connection, the SAS System is in a wait state. Each time a new connection is established, the EOV= variable in the DATA step is set to 1. Because the server accepts only one connection at a time, no new connections can be established until the current connection is closed. The connection closes automatically when the remote client application disconnects. The SOCKET access method continues to accept new connections until it reaches the limit set in the RECONN option.

Examples

Example 1: Communicating between Two SAS Applications over a TCP/IP Socket This example shows how two SAS applications can talk over a TCP/IP socket. The local application is in server mode; the remote application is the client that connects to the server. This example assumes that the server host name is **hp720.unx.sas.com**, that the well-known port number is 5000, and that the server allows a maximum of three connections before closing the socket.

Here is the program for the server application:

```
filename local socket ':5000' server reconn=3;
   /*The server is using a reserved */
   /*port number of 5000.           */

data tcpip;
   infile local eov=v;
   input x $10;
   if v=1 then
      do;              /* new connection when v=1 */
         put 'new connection received';
      end;
   output;
run;
```

Here is the program for the remote client application:

```
filename remote socket 'hp720.unx.sas.com:5000';

data _null_;
   file remote;
   do i=1 to 10;
      put i;
   end;
run;
```

See Also

Statements:

"FILENAME" on page 821
"FILENAME, CATALOG Access Method" on page 826
"FILENAME, FTP Access Method" on page 829

FILENAME, URL Access Method

Allows you to access remote files using the URL access method

Valid: anywhere

Category: Data Access

Syntax

FILENAME *fileref* URL *'external-file'*
<*url-options*>;

Arguments

fileref
is a valid fileref.

Tip: The association between a fileref and an external file lasts only for the duration of the SAS session or until you change it or discontinue it with another FILENAME statement. You can change the fileref for a file as often as you want.

URL
specifies the access method that enables you to read a file from any host machine that you can connect to on a network with a URL server running.

Alias: HTTP

'external-file'
specifies the name of the file you want to read from on a URL server. The access method must be in one of these forms

http://*hostname*/*file*

http://*hostname*:*portno*/*file*

Operating Environment Information: For details on specifying the physical names of external files, see the SAS documentation for your operating environment. △

url-options
can be any of the following:

BLOCKSIZE=*blocksize*
where *blocksize* is the size of the URL data buffer in bytes.

Default: 8K

DEBUG
writes debugging information to the SAS log.

Tip: The result of the HELP command is returned as records.

LRECL=*lrecl*
where *lrecl* is the logical record length.

Default: 256

PASS='*password*'
where *password* is the password to use with the user name that is specified in the USER option.

Tip: You can specify the PROMPT option instead of the PASS option, which tells the system to prompt you for the password.

PROMPT
specifies to prompt for the user login password if necessary.

Tip: If you specify PROMPT, you do not need to specify PASS=.

PROXY=*url*
specifies the universal resource locator (URL) for the proxy server in one of these forms:

http://*hostname*/

http://*hostname:portno*/

RECFM=*recfm*
where *recfm* is one of three record formats:

F is fixed record format. Thus, all records are of size LRECL with no line delimiters. Data are transferred in image (binary) mode.

S is stream record format. Data are transferred in image (binary) mode.

Tip: The amount of data that is read is controlled by the current LRECL value or the value of the NBYTE= variable in the FILE or INFILE statement. The NBYTE= option specifies a variable equal to the amount of data to be read. This amount must be less than or equal to LRECL.

See Also: The NBYTE= option on page 808 in the FILE statement and the NBYTE= option on page 862 in the INFILE statement.

V is variable record format (the default). In this format, records have varying lengths, and they are transferred in text (stream) mode.

Tip: Any record larger than LRECL is truncated.

Default: V

USER='*username*' where *username* is used to log in to the URL server.

Tip: If you specify **user='*'**, then the user is prompted for an id.

Interaction: If PROMPT is specified, but USER= is not, the user is prompted for an id as well as a password.

Details

Operating Environment Information: Using the FILENAME statement requires information specific to your operating environment. The URL access method is fully documented here, but for more information on how to specify file names, see the SAS documentation for your operating environment. △

Examples

Example 1: Accessing a File at a Web Site This example accesses document **test.dat** at site **www.a.com**:

```
filename foo url 'http://www.a.com/test.dat'
         proxy='http://www.gt.sas.com';
```

Example 2: Specifying a User Id and a Password This example accesses document **file1.html** at site **www.b.com** and requires a user id and password:

```
filename foo url 'http://www.b.com/file1.html'
         user='jones' prompt;
```

Example 3: Reading the First 15 Records from a URL File This example reads the first 15 records from a URL file and writes them to the SAS log with a PUT statement:

```
filename foo url
    'http://www.sas.com/service/techsup/intro.html';

data _null_;
   infile foo length=len;
   input record $varying200. len;
   put record $varying200. len;
   if _n_=15 the stop;
run;
```

See Also

Statements:

"FILENAME" on page 821
"FILENAME, CATALOG Access Method" on page 826
"FILENAME, FTP Access Method" on page 829
"FILENAME, SOCKET Access Method" on page 835

FOOTNOTE

Prints up to ten lines of text at the bottom of the procedure or DATA step output

Valid: anywhere
Category: Output Control
Requirement: You must specify the FOOTNOTE option if you use a FILE statement.

Syntax

FOOTNOTE<*n* > <'*text*' | "*text*" >;

Without Arguments

Using FOOTNOTE without arguments cancels all existing footnotes.

Arguments

n
specifies the relative line to be occupied by the footnote.

Tip: For footnotes, lines are pushed up from the bottom. The FOOTNOTE statement with the highest number appears on the bottom line.

Range: *n* can range from 1 to 10.

Default: If you omit *n*, SAS assumes a value of 1.

'*text*' | "*text*"
specifies the text of the footnote in single or double quotation marks.

Tip: For compatibility with previous releases, SAS accepts some text without quotation marks. When you write new programs or update existing programs, *always* surround text with quotation marks.

Details

A FOOTNOTE statement takes effect when the step or RUN group with which it is associated executes. Once you specify a footnote for a line, SAS repeats the same footnote on all pages until you cancel or redefine the footnote for that line. When a FOOTNOTE statement is specified for a given line, it cancels the previous FOOTNOTE statement for that line and for all footnote lines with higher numbers.

Operating Environment Information: The maximum footnote length allowed depends on the operating environment and the value of the LINESIZE= system option. Refer to the SAS documentation for your operating environment for more information. △

Comparisons

You can also create footnotes with the FOOTNOTES window. For more information, refer to the online help for the window.

Examples

These examples of a FOOTNOTE statement result in the same footnote:

```
□ footnote8 "Managers' Meeting";

□ footnote8 'Managers'' Meeting';
```

See Also

Statement:

"TITLE" on page 1020

FORMAT

Associates formats with variables

Valid: in a DATA step or PROC step
Category: Information
Type: Declarative

Syntax

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

Arguments

variable
names one or more variables for SAS to associate with a format. You must specify at least one *variable*.

Tip: To disassociate a format from a variable, use the variable in a FORMAT statement without specifying a format in a DATA step or in PROC DATASETS. In a DATA step, place this FORMAT statement after the SET statement. See Example 2 on page 845. You can also use PROC DATASETS.

format
specifies the format that is listed for writing the values of the variables.

Tip: Formats that are associated with variables by using a FORMAT statement behave like formats that are used with a colon modifier in a subsequent PUT statement. For details on using a colon modifer, see "PUT, List" on page 983.

See also: "Formats by Category" on page 66

DEFAULT=*default-format*
specifies a temporary default format for displaying the values of variables that are listed in the FORMAT statement. These default formats apply only to the current DATA step; they are not permanently associated with variables in the output data set.
A DEFAULT= format specification applies to

- variables that are not named in a FORMAT or ATTRIB statement
- variables that are not permanently associated with a format within a SAS data set
- variables that are not written with the explicit use of a format.

Default: If you omit DEFAULT=, SAS uses BEST*w*. as the default numeric format and $*w*. as the default character format.

Tip: A DEFAULT= specification can occur anywhere in a FORMAT statement. It can specify either a numeric default, a character default, or both.

Valid: in a DATA step

Featured in: Example 1 on page 844

Details

The FORMAT statement can use standard SAS formats or user-written formats that have been previously defined in PROC FORMAT. A single FORMAT statement can associate the same format with several variables, or it can associate different formats with different variables. If a variable appears in multiple FORMAT statements, SAS uses the format that is assigned last.

You use a FORMAT statement in the DATA step to permanently associate a format with a variable. SAS changes the descriptor information of the SAS data set that contains the variable. You can use a FORMAT statement in some PROC steps, but the rules are different. For more information, see *SAS Procedures Guide*.

Comparisons

Both the ATTRIB and FORMAT statements can associate formats with variables, and both statements can change the format that is associated with a variable. You can use the FORMAT statement in PROC DATASETS to change or remove the format that is associated with a variable. You can also associate, change, or disassociate formats and variables in existing SAS data sets through the windowing environment.

Examples

Example 1: Assigning Formats and Defaults This example uses a FORMAT statement to assign formats and default formats for numeric and character variables. The default formats are not associated with variables in the data set but affect how the PUT statement writes the variables in the current DATA step.

```
data tstfmt;
   format W $char3.
          Y 10.3
          default=8.2 $char8.;
   W='Good morning.';
   X=12.1;
   Y=13.2;
   Z='Howdy-doody';
   put W/X/Y/Z;
run;

proc contents data=tstfmt;
run;

proc print data=tstfmt;
run;
```

The following output shows a partial listing from PROC CONTENTS, as well as the report that PROC PRINT generates.

```
                          The SAS System                          3

                       CONTENTS PROCEDURE

      -----Alphabetic List of Variables and Attributes-----

         #    Variable    Type    Len    Pos    Format
         ------------------------------------------------
         1    W           Char      3     16    $CHAR3.
         3    X           Num       8      8
         2    Y           Num       8      0    10.3
         4    Z           Char     11     19
```

```
                          The SAS System                          4

        OBS    W              Y       X          Z

         1     Goo         13.200    12.1    Howdy-doody
```

The default formats apply to variables X and Z while the assigned formats apply to the variables W and Y.

The PUT statement produces this result:

```
----+----1----+----2
Goo
   12.10
3.200
Howdy-do
```

Example 2: Removing a Format This example disassociates an existing format from a variable in a SAS data set. The order of the FORMAT and the SET statements is important.

```
data rtest;
   set rtest;
   format x;
run;
```

See Also

Statement:

"ATTRIB" on page 762

The DATASETS Procedure in *SAS Procedures Guide*

GO TO

Moves execution immediately to the statement label that is specified

Valid: in a DATA step

Category: Control

Type: Executable

Alias: GOTO

Syntax

GO TO *label*;

Arguments

label
specifies a statement label that identifies the GO TO destination. The destination must be within the same DATA step. You must specify the *label* argument.

Comparisons

The GO TO statement and the LINK statement are similar. However, a GO TO statement is often used without a RETURN statement, whereas a LINK statement is usually used with an explicit RETURN statement. The action of a subsequent RETURN statement differs between the GO TO and LINK statements. A RETURN statement after a LINK statement returns execution to the statement that follows the LINK statement. A RETURN after a GO TO statement returns execution to the beginning of the DATA step (unless a LINK statement precedes the GO TO statement, in which case execution continues with the first statement after the LINK statement).

GO TO statements can often be replaced by DO-END and IF-THEN/ELSE programming logic.

Examples

Use the GO TO statement as shown here.

- In this example, if the condition is true, the GO TO statement instructs SAS to jump to a label called ADD and to continue execution from there. If the condition is false, SAS executes the PUT statement and the statement that is associated with the GO TO label:

```
data info;
   input x;
   if 1<=x<=5 then go to add;
   put x=;
   add: sumx+x;
   datalines;
7
6
323
;
```

  Because every DATA step contains an implied RETURN at the end of the step, program execution returns to the top of the step after the sum statement is executed. Therefore, an explicit RETURN statement at the bottom of the DATA step is not necessary.

- If you do not want the sum statement to execute for observations that do not meet the condition, rewrite the code and include an explicit return statement.

```
data info;
   input x;
```

```
   if 1<=x<=5 then go to add;
   put x=;
   return;
      /* SUM statement not executed */
      /* if x<1 or x>5              */
   add: sumx+x;
   datalines;
7
6
323
;
```

See Also

Statements:

"DO" on page 789
"Labels, Statement" on page 908
"LINK" on page 923
"RETURN" on page 1004

IF, Subsetting

Continues processing only those observations that meet the condition

Valid: in a DATA step
Category: Action
Type: Executable

Syntax

IF *expression*;

Arguments

expression
is any SAS expression.

Details

The subsetting IF statement causes the DATA step to continue processing only those raw data records or those observations from a SAS data set that meet the condition of the expression that is specified in the IF statement. That is, if the expression is true for the observation or record (its value is neither 0 nor missing), SAS continues to execute statements in the DATA step and includes the current observation in the data set. The resulting SAS data set or data sets contain a subset of the original external file or SAS data set.

If the expression is false (its value is 0 or missing), no further statements are processed for that observation or record, the current observation is not written to the

data set, and the remaining program statements in the DATA step are not executed. SAS immediately returns to the beginning of the DATA step because the subsetting IF statement does not require additional statements to stop processing observations.

Comparisons

- The subsetting IF statement is equivalent to this IF-THEN statement:

```
if not (expression)
   then delete;
```

- When you create SAS data sets, use the subsetting IF statement when it is easier to specify a condition for including observations. When it is easier to specify a condition for excluding observations, use the DELETE statement.
- The subsetting IF and the WHERE statements are not equivalent. The two statements work differently and produce different output data sets in some cases. The most important differences are summarized as follows:
  - The subsetting IF statement selects observations that have been read into the program data vector. The WHERE statement selects observations before they are brought into the program data vector. The subsetting IF might be less efficient than the WHERE statement because it must read each observation from the input data set into the program data vector.
  - The subsetting IF statement and WHERE statement can produce different results in DATA steps that interleave, merge, or update SAS data sets.
  - When the subsetting IF statement is used with the MERGE statement, the SAS System selects observations after the current observations are combined. When the WHERE statement is used with the MERGE statement, the SAS System applies the selection criteria to each input data set before combining the current observations.
  - The subsetting IF statement can select observations from an existing SAS data set or from raw data that are read with the INPUT statement. The WHERE statement can select observations only from existing SAS data sets.
  - The subsetting IF statement is executable; the WHERE statement is not.

Examples

- This example results in a data set that contains only those observations with the value **F** for the variable SEX:

```
if sex='F';
```

- This example results in a data set that contains all observations for which the value of the variable AGE is not missing or 0:

```
if age;
```

See Also

Data Set Options:

"WHERE=" on page 43

Statements:

"DELETE" on page 783
"IF-THEN/ELSE" on page 849
"WHERE" on page 1028

"WHERE Processing" in *SAS Language Reference: Concepts*

IF-THEN/ELSE

Executes a SAS statement for observations that meet specific conditions

Valid: in a DATA step
Category: Control
Type: Executable

Syntax

IF *expression* **THEN** *statement*;
<**ELSE** *statement*;>

Arguments

expression
is any SAS expression and is a required argument.

statement
can be any executable SAS statement or DO group.

Details

SAS evaluates the expression in an IF-THEN statement to produce a result that is either nonzero, zero, or missing. A nonzero and nonmissing result causes the expression to be true; a result of zero or missing causes the expression to be false.

If the conditions that are specified in the IF clause are met, the IF-THEN statement executes a SAS statement for observations that are read from a SAS data set, for records in an external file, or for computed values. An optional ELSE statement gives an alternative action if the THEN clause is not executed. The ELSE statement, if used, must immediately follow the IF-THEN statement.

Using IF-THEN statements *without* the ELSE statement causes SAS to evaluate all IF-THEN statements. Using IF-THEN statements *with* the ELSE statement causes SAS to execute IF-THEN statements until it encounters the first true statement. Subsequent IF-THEN statements are not evaluated.

Note: For greater efficiency, construct your IF-THEN/ELSE statement with conditions of decreasing probability. △

Comparisons

- Use a SELECT group rather than a series of IF-THEN statements when you have a long series of mutually exclusive conditions.
- Use subsetting IF statements, without a THEN clause, to continue processing only those observations or records that meet the condition that is specified in the IF clause.

Examples

These examples show different ways of specifying the IF-THEN/ELSE statement.

□

```
if x then delete;
```

□

```
if status='OK' and type=3 then count+1;
```

□

```
if age ne agecheck then delete;
```

□

```
if x=0 then
   if y ne 0 then put 'X ZERO, Y NONZERO';
   else put 'X ZERO, Y ZERO';
else put 'X NONZERO';
```

□

```
if answer=9 then
   do;
      answer=.;
      put 'INVALID ANSWER FOR ' id=;
   end;
else
   do;
      answer=answer10;
      valid+1;
   end;
```

□

```
data region;
   input city $ 1-30;
   if city='New York City'
      or city='Miami' then
      region='ATLANTIC COAST';
   else if city='San Francisco'
      or city='Los Angeles' then
         region='PACIFIC COAST';
   datalines;
...more data lines...
;
```

See Also

Statements:

"DO" on page 789
"IF, Subsetting" on page 847
"SELECT" on page 1007

%INCLUDE

Brings a SAS programming statement, data lines, or both, into a current SAS program

Valid: anywhere
Category: Program Control
Alias: %INC

Syntax

%INCLUDE *source(s)*
</<SOURCE2> <S2=*length*> <*host-options*>>;

Arguments

source
describes the location of the information that you want to access with the %INCLUDE statement. There are three possible sources:

Source	Definition
file-specification	specifies an external file
internal-lines	specifies lines that are entered earlier in the same SAS job or session
keyboard-entry	specifies statements or data lines that you enter directly from the terminal

file-specification
identifies an entire external file that you want to bring into your program.

Restriction: You cannot selectively include lines from an external file.

Tip: Including external sources is useful in all types of SAS processing: batch, windowing, interactive line, and noninteractive.

Operating Environment Information: For complete details on specifying the physical names of external files, see the SAS documentation for your operating environment. △

File-specification can have these forms:

'external-file'
specifies the physical name of an external file that is enclosed in quotation marks. The physical name is the name by which the operating environment recognizes the file.

fileref
specifies a fileref that has previously been associated with an external file.

Tip: You can use a FILENAME statement or function or an operating environment command to make the association.

fileref(filename-1 <, "filename–2.xxx", ... filename-n>)
specifies a fileref that has previously been associated with an aggregate storage location. Follow the fileref with one or more filenames that reside in that location. Enclose the filenames in one set of parentheses, and separate each filename with a comma, space.

This example instructs SAS to include the files "testcode1.sas", "testcode2.sas" and "testcode3.txt." These files are located in aggregate storage location "mycode." You do not need to specify the file extension for testcode1 and testcode2 because they are the default .SAS extension. You must enclose testcode3.txt in quotation marks with the whole filename specified because it has an extension other than .SAS:

```
%include mylib(testcode1, testcode2,
               "testcode3.txt");
```

Tip: You can use a FILENAME statement or function or an operating environment command to make the association.

Operating Environment Information: Different operating environments call an aggregate grouping of files by different names, such as a directory, a MACLIB, a text library, or a partitioned data set. For complete details on specifying external files, see the SAS documentation for your operating environment. △

Operating Environment Information: The character length that is allowed for filenames is operating environment specific. For information on accessing files from a storage location that contains several files, see the SAS documentation for your operating environment. △

internal-lines
includes lines that are entered earlier in the same SAS job or session.

To include internal lines, use any of the following:

n	includes line *n*.
n-m or *n:m*	includes lines *n* through *m*.

Tip: Including internal lines is most useful in interactive line mode processing.

Tip: Use a %LIST statement to determine the line numbers that you want to include.

Tip: Although you can use the %INCLUDE statement to access previously submitted lines when you run SAS in a windowing environment, it may be more practical to recall lines in the Program Editor with the RECALL command and then submit the lines with the SUBMIT command.

Note: The SPOOL system option controls internal access to previously submitted lines when you run SAS in interactive line mode, noninteractive mode,

and batch mode. Use the OPTIONS procedure to determine the current setting of the SPOOL system option on your system. △

keyboard-entry
is a method for preparing a program so that you can interrupt the current program's execution, enter statements or data lines from the keyboard, and then resume program processing.

Tip: Use this method when you run SAS in noninteractive or interactive line mode. SAS pauses during processing and prompts you to enter statements from the keyboard.

Tip: Use this argument to include source from the keyboard:

* prompts you to enter data from the terminal. Place an asterisk (*) after the %INCLUDE statement in your code:

```
proc print;
   %include *;
run;
```

To resume processing the original source program, enter a %RUN statement from the terminal.

Tip: You can use a %INCLUDE * statement in a batch job by creating a file with the fileref SASTERM that contains the statements that you would otherwise enter from the terminal. The %INCLUDE * statement causes SAS to read from the file that is referenced by SASTERM. Insert a %RUN statement into the file that is referenced by SASTERM where you want SAS to resume reading from the original source.

Note: The fileref SASTERM must have been previously associated with an external file in a FILENAME statement or function or an operating environment command. △

SOURCE2
causes the SAS log to show the source statements that are being included in your SAS program.

Tip: The SAS log also displays the fileref and the filename of the source and the level of nesting (1, 2, 3, and so on).

Tip: The SAS system option SOURCE2 produces the same results. When you specify SOURCE2 in a %INCLUDE statement, it overrides the setting of the SOURCE2 system option for the duration of the include operation.

S2=*length*
specifies the length of the record to be used for input. *Length* can have these values:

S sets S2 equal to the current setting of the S= SAS system option.

0 tells SAS to use the setting of the SEQ= system option to determine whether the line contains a sequence field. If the line does contain a sequence field, SAS determines line length by excluding the sequence field from the total length.

n specifies a number greater than zero that corresponds to the length of the line to be read, when the file contains fixed-length records. When the file contains variable-length records, *n* specifies the column in which to begin reading data.

Tip: Text input from the %INCLUDE statement can be either fixed or variable length.

- Fixed-length records are either unsequenced or sequenced at the end of each record. For fixed-length records, the value given in S2= is the ending column of the data.
- Variable-length records are either unsequenced or sequenced at the beginning of each record. For variable-length records, the value given in S2= is the starting column of the data.

Interaction: The S2= system option also specifies the length of secondary source statements that are accessed by the %INCLUDE statement, and it is effective for the duration of your SAS session. The S2= option in the %INCLUDE statement affects only the current include operation. If you use the option in the %INCLUDE statement, it overrides the system option setting for the duration of the include operation.

See Also: For a detailed discussion of fixed- and variable-length input records, see the S= system option in "S=" on page 1146, and the S2= system option in "S2=" on page 1148.

host-options

Operating Environment Information: Operating environments can support various options for the %INCLUDE statement. See the documentation for your operating environment for a list of these options and their functions. △

Details

What %INCLUDE Does When you execute a program that contains the %INCLUDE statement, SAS executes your code, including any statements or data lines that you bring into the program with %INCLUDE.

Operating Environment Information: Use of the %INCLUDE statement is dependent on your operating environment. See the documentation for your operating environment for more information about additional software features and methods of referring to and accessing your files. See your documentation before you run the examples for this statement. △

Three Sources of Data The %INCLUDE statement accesses SAS statements and data lines from three possible sources:

- external files
- lines entered earlier in the same job or session
- lines entered from the keyboard.

When Useful The %INCLUDE statement is most often used when running SAS in interactive line mode, noninteractive mode, or batch mode. Although you can use the %INCLUDE statement when you run SAS using windows, it may be more practical to use the INCLUDE and RECALL commands to access data lines and program statements, and submit these lines again.

Rules for Using %INCLUDE

- You can specify any number of sources in a %INCLUDE statement, and you can mix the types of included sources. Note, however, that although it is possible to include information from multiple sources in one %INCLUDE statement, it might be easier to understand a program that uses separately coded %INCLUDE statements for each source.

- The %INCLUDE statement must begin at a statement boundary. That is, it must be the first statement in a SAS job or must immediately follow a semicolon ending another statement. A %INCLUDE statement cannot immediately follow a DATALINES, DATALINES4, CARDS, or CARDS4 statement (or PARMCARDS or PARMCARDS4 statement in procedures that use those statements); however, you can include data lines with the %INCLUDE statement using one of these methods:
  - Make the DATALINES, DATALINES4, or CARDS, CARDS4 statement the first line in the file that contains the data.
  - Place the DATALINES, DATALINES4, or CARDS, CARDS4 statement in one file, and the data lines in another file. Use both sources in a single %INCLUDE statement.

  The %INCLUDE statement can be nested within a file that has been accessed with %INCLUDE. The maximum number of nested %INCLUDE statements that you can use depends on system-specific limitations of your operating environment (such as available memory or the number of files you can have open concurrently).

Comparisons

The %INCLUDE statement executes statements immediately. The INCLUDE command brings the included lines into the PROGRAM EDITOR window but does not execute them. You must issue the SUBMIT command to execute those lines.

Examples

Example 1: Including an External File

- This example stores a portion of a program in a file and includes it in a program to be written later. This program is stored in a file named MYFILE:

```
data monthly;
   input x y month $;
   datalines;
1 1 January
2 2 February
3 3 March
4 4 April
;
```

  This program includes an external file named MYFILE and submits the DATA step that it contains before the PROC PRINT step executes:

```
%include 'MYFILE';

proc print;
run;
```

- To reference a file by using a fileref rather than the actual file name, you can use the FILENAME statement (or a command recognized by your operating environment) to assign a fileref:

```
filename in1 'MYFILE';
```

  You can later access MYFILE with the fileref IN1:

```
%inc in1;
```

- If you want to use many files that are stored in a directory, PDS, or MACLIB (or whatever your operating environment calls an aggregate storage location), you can

assign the fileref to the larger storage unit and then specify the filename. For example, this FILENAME statement assigns the fileref STORAGE to an aggregate storage location:

```
filename storage
   'aggregate-storage-location';
```

You can later include a file using this statement:

```
%inc storage(MYFILE);
```

- You can also access several files or members from this storage location by listing them in parentheses after the fileref in a single %INCLUDE statement. Separate filenames with a comma or a blank space. The following %INCLUDE statement demonstrates this method:

```
%inc storage(file-1,file-2,file-3);
```

When the file does not have the default .SAS extension, you can access it using quotation marks around the complete filename listed inside the parentheses.

-

```
%inc storage("file-1.txt","file-2.dat",
             "file-3.cat");
```

Example 2: Including Previously Submitted Lines

This %INCLUDE statement causes SAS to process lines 1, 5, 9 through 12, and 13 through 16 as though you had entered them again from your keyboard:

```
%include 1 5 9-12 13:16;
```

Example 3: Including Input from the Keyboard

CAUTION:
The method shown in this example is valid only when you run SAS in noninteractive mode or interactive line mode. △

This example uses %INCLUDE to add a customized TITLE statement when PROC PRINT executes:

```
  data report;
   infile file-specification;
   input month $ salesamt $;
run;

proc print;
   %include *;
run;
```

When this DATA step executes, %INCLUDE with the asterisk causes SAS to issue a prompt for statements that are entered at the terminal. You can enter statements such as

```
where month= 'January';

title 'Data for month of January';
```

After you enter statements, you can use %RUN to resume processing by typing

```
%run;
```

The %RUN statement signals to SAS to leave keyboard-entry mode and resume reading and executing the remaining SAS statements from the original program.

Example 4: Using %INCLUDE with Several Entries in a Single Catalog This example submits the source code from three entries in the catalog MYLIB.INCLUDE. When no entry type is specified, the default is CATAMS.

```
filename dir catalog 'mylib.include';
%include dir(mem1);
%include dir(mem2);
%include dir(mem3);
```

See Also

Statements:

"%LIST" on page 926
"%RUN" on page 1006

INFILE

Identifies an external file to read with an INPUT statement

Valid: in a DATA Step
Category: File-handling
Type: Executable

Syntax

INFILE *file-specification* <*options*> <*host-options*>;

INFILE *DBMS-specifications;*

Arguments

file-specification
identifies the source of the input data records, which is an external file or in-stream data. *File-specification* can have these forms:

'external-file'
specifies the physical name of an external file. The physical name is the name that the operating environment uses to access the file.

fileref
specifies the fileref of an external file.

Requirement: You must have previously associated the fileref with an external file in a FILENAME statement, FILENAME function, or an appropriate operating environment command.

See: "FILENAME" on page 821

fileref(file)
specifies a fileref of an aggregate storage location and the name of a file or member, enclosed in parentheses, that resides in that location.

Operating Environment Information: Different operating environments call an aggregate grouping of files by different names, such as a directory, a MACLIB, or a partitioned data set. For details on how to specify external files, see the SAS documentation for your operating environment. △

Requirement: You must have previously associated the fileref with an external file in a FILENAME statement, a FILENAME function, or an appropriate operating environment command.

See: "FILENAME" on page 821

CARDS | CARDS4
for a definition, see DATALINES.

Alias: DATALINES | DATALINES4

DATALINES | DATALINES4
specifies that the input data immediately follows the DATALINES or DATALINES4 statement in the DATA step. This allows you to use the INFILE statement options to control how the INPUT statement reads instream data lines.

Alias: CARDS | CARDS4

Featured in: Example 1 on page 868

Options

BLKSIZE=*block-size*
specifies the block size of the input file.

Default: dependent on the operating environment

Operating Environment Information: For details, see the SAS documentation for your operating environment. △

COLUMN=*variable*
names a variable that SAS uses to assign the current column location of the input pointer. Like automatic variables, the COLUMN= variable is not written to the data set.

Alias: COL=

See Also: LINE= on page 861

Featured in: Example 7 on page 872

DELIMITER=*delimiter(s)*
specifies a delimiter for list input, where *delimiter* is

'*list-of-delimiting-characters*'
specifies one or more characters to read as delimiters.

Requirement: Enclose the list of characters in quotation marks.

Featured in: Example 1 on page 868

character-variable
specifies a character variable whose value becomes the delimiter.

Alias: DLM=

Default: blank space

Tip: DELIMITER= allows you to use list input even when the data are separated by characters other than spaces.

See: "Reading Delimited Data" on page 866

See Also: DSD option on page 859

Featured in: Example 1 on page 868

DSD
changes how SAS treats delimiters when list input is used and sets the default delimiter to a comma. When you specify DSD, SAS treats two consecutive delimiters as a missing value and removes quotation marks from character values.

Interaction: Use the DELIMITER= option to change the delimiter.

Tip: Use the DSD option and list input to read a character value that contains a delimiter within a quoted string. The INPUT statement treats the delimiter as a valid character and removes the quotation marks from the character string before the value is stored. Use the tilde (~) format modifier to retain the quotation marks.

See: "Reading Delimited Data" on page 866

See Also: DELIMITER= on page 858

Featured in: Example 2 on page 869 and Example 1 on page 868

END=*variable*
names a variable that SAS sets to 1 when the current input data record is the last in the input file. Until SAS processes the last data record, the END= variable is set to 0. Like automatic variables, this variable is not written to the data set.

Restriction: You cannot use the END= option with

- the UNBUFFERED option
- the DATALINES or DATALINES4 statement
- an INPUT statement that reads multiple input data records.

Tip: Use the option EOF= on page 859 when END= is invalid.

Featured in: Example 4 on page 870

EOF=*label*
specifies a statement label that is the object of an implicit GO TO when the INFILE statement reaches end-of-file. When an INPUT statement attempts to read from a file that has no more records, SAS moves execution to the statement label indicated.

Interaction: Use EOF= instead of the END= option with

- the UNBUFFERED option
- the DATALINES or DATALINES4 statement
- an INPUT statement that reads multiple input data records.

Tip: The EOF= option is useful when you read from multiple input files sequentially.

See Also: END= on page 859, EOV= on page 859, and UNBUFFERED on page 864

EOV=*variable*
names a variable that SAS sets to 1 when the first record in a file in a series of concatenated files is read. The variable is set only after SAS encounters the next file. Like automatic variables, the EOV= variable is not written to the data set.

Tip: Reset the EOV= variable back to 0 after SAS encounters each boundary.

See Also: END= on page 859 and EOF= on page 859

EXPANDTABS | NOEXPANDTABS
specifies whether to expand tab characters to the standard tab setting, which is set at 8-column intervals that start at column 9.

Default: NOEXPANDTABS

Tip: EXPANDTABS is useful when you read data that contain the tab character that is native to your operating environment.

FILENAME=*variable*
names a variable that SAS sets to the physical name of the currently opened input file. Like automatic variables, the FILENAME= variable is not written to the data set.

Tip: Use a LENGTH statement to make the variable length long enough to contain the value of the filename.

See Also: FILEVAR= on page 860

Featured in: Example 4 on page 870

FILEVAR=*variable*
names a variable whose change in value causes the INFILE statement to close the current input file and open a new one. When the next INPUT statement executes, it reads from the new file that the FILEVAR= variable specifies. Like automatic variables, this variable is not written to the data set.

Restriction: The FILEVAR= variable must contain a character string that is a physical filename.

Interaction: When you use the FILEVAR= option, the *file-specification* is just a placeholder, not an actual filename or a fileref that has been previously-assigned to a file. SAS uses this placeholder for reporting processing information to the SAS log. It must conform to the same rules as a fileref.

Tip: Use FILEVAR= to dynamically change the currently opened input file to a new physical file.

See Also: "Updating External Files in Place" on page 865

Featured in: Example 4 on page 870

FIRSTOBS=*record-number*
specifies a record number that SAS uses to begin reading input data records in the input file.

Default: 1

Tip: Use FIRSTOBS= with OBS= to read a range of records from the middle of a file.

Example: This statement processes record 50 through record 100:

```
infile file-specification firstobs=50 obs=100;
```

FLOWOVER
causes an INPUT statement to continue to read the next input data record if it does not find values in the current input line for all the variables in the statement. This is the default behavior of the INPUT statement.

See: "Reading Past the End of a Line" on page 867

See Also: MISSOVER on page 861, STOPOVER on page 863, and TRUNCOVER on page 863

LENGTH=*variable*
names a variable that SAS sets to the length of the current input line. SAS does not assign the variable a value until an INPUT statement executes. Like automatic variables, the LENGTH= variable is not written to the data set.

Tip: This option in conjunction with the $VARYING informat on page 657 is useful when the field width varies.

Featured in: Example 3 on page 870 and Example 6 on page 872

LINE=*variable*
names a variable that SAS sets to the line location of the input pointer in the input buffer. Like automatic variables, the LINE= variable is not written to the data set.

Range: 1 to the value of the N= option

Interaction: The value of the LINE= variable is the current relative line number within the group of lines that is specified by the N= option or by the #*n* line pointer control in the INPUT statement.

See Also: COLUMN= on page 858 and N= on page 862

Featured in: Example 7 on page 872

LINESIZE=*line-size*
specifies the record length that is available to the INPUT statement.

Operating Environment Information: Values for *line-size* are dependent on the operating environment record size. For details, see the SAS documentation for your operating environment. △

Alias: LS=

Range: up to 32767

Interaction: If an INPUT statement attempts to read past the column that is specified by the LINESIZE= option, the action that is taken depends on whether the FLOWOVER, MISSOVER, SCANOVER, STOPOVER, or TRUNCOVER option is in effect. FLOWOVER is the default.

Tip: Use LINESIZE= to limit the record length when you do not want to read the entire record.

Example: If your data lines contain a sequence number in columns 73 through 80, use this INFILE statement to restrict the INPUT statement to the first 72 columns:

```
infile file-specification linesize=72;
```

LRECL=*logical-record-length*
specifies the logical record length.

Operating Environment Information: Values for *logical-record-length* are dependent on the operating environment. For details, see the SAS documentation for your operating environment. △

Default: dependent on the operating environment's file characteristics.

Tip: LRECL= specifies the physical line length of the file. LINESIZE= tells the INPUT statement how much of the line to read.

MISSOVER
prevents an INPUT statement from reading a new input data record if it does not find values in the current input line for all the variables in the statement. When an INPUT statement reaches the end of the current input data record, variables without any values assigned are set to missing.

Tip: Use MISSOVER if the last field(s) may be missing and you want SAS to assign missing values to the corresponding variable.

See: "Reading Past the End of a Line" on page 867

See Also: FLOWOVER on page 860, SCANOVER on page 863, STOPOVER on page 863, and TRUNCOVER on page 863

Featured in: Example 2 on page 869

N=*available-lines*
specifies the number of lines that are available to the input pointer at one time.

Default: the highest value following a # pointer control in any INPUT statement in the DATA step. If you omit a # pointer control, the default value is 1.

Interaction: This option affects only the number of lines that the pointer can access at a time; it has no effect on the number of lines an INPUT statement reads.

Tip: When you use # pointer controls in an INPUT statement that are less than the value of N=, you might get unexpected results. To prevent this, include a # pointer control that equals the value of the N= option. For example,

```
infile 'external file' n=5;
input #2 name : $25. #3 job : $25. #5;
```

The INPUT statement includes a #5 pointer control, even though no data are read from that record.

Featured in: Example 7 on page 872

NBYTE=*variable*
specifies the name of a variable that contains the number of bytes to read from a file when you are reading data in stream record format (RECFM=S in the FILENAME statement).

Default: the LRECL value of the file

Interaction: If the number of bytes to read is set to -1, the FTP and SOCKET access methods return the number of bytes that are currently available in the input buffer.

See: the FILENAME, SOCKET RECFM= option on page 836 and the FILENAME, FTP RECFM= option on page 832

OBS=*record-number*
specifies the record number of the last record to read in an input file that is read sequentially.

Tip: Use OBS= with FIRSTOBS= to read a range of records from the middle of a file.

Example: This statement processes only the first 100 records in the file:

```
infile file-specification obs=100;
```

PAD | NOPAD
controls whether SAS pads the records that are read from an external file with blanks to the length that is specified in the LRECL= option.

Default: NOPAD

See Also: LRECL= on page 861

PRINT | NOPRINT
specifies whether the input file contains carriage control characters.

Tip: To read a print file in a DATA step without having to remove the carriage control characters, specify PRINT. To read the carriage control characters as data values, specify NOPRINT.

RECFM=*record-format*
specifies the record format of the input file.

Operating Environment Information: Values for *record-format* are dependent on the operating environment. For details, see the SAS documentation for your operating environment. △

SCANOVER
causes the INPUT statement to scan the input data records until the character string that is specified in the @'*character-string*' expression is found.

Interaction: The MISSOVER, TRUNCOVER, and STOPOVER options change how the INPUT statement behaves when it scans for the @'*character-string*' expression and reaches the end of record. By default (FLOWOVER option), the INPUT statement scans the next record while these other options cause scanning to stop.

Tip: It is redundant to specify both SCANOVER and FLOWOVER.

See: "Reading Past the End of a Line" on page 867

See Also: FLOWOVER on page 860, MISSOVER on page 861, STOPOVER on page 863, and TRUNCOVER on page 863

SHAREBUFFERS
specifies that the FILE statement and the INFILE statement share the same buffer.

Alias: SHAREBUFS

Tip: Use SHAREBUFFERS with the INFILE, FILE, and PUT statements to update an external file in place. This saves CPU time because the PUT statement output is written straight from the input buffer instead of the output buffer.

Tip: Use SHAREBUFFERS to update specific fields in an external file instead of an entire record.

Featured in: Example 5 on page 871

START=*variable*
names a variable whose value SAS uses as the first column number of the record that the PUT _INFILE_ statement writes. Like automatic variables, the START variable is not written to the data set.

See Also: _INFILE_ option on page 963 in the PUT statement

STOPOVER
causes the DATA step to stop processing if an INPUT statement reaches the end of the current record without finding values for all variables in the statement. When an input line does not contain the expected number of values, SAS sets _ERROR_ to 1, stops building the data set as though a STOP statement has executed, and prints the incomplete data line.

Tip: Use FLOWOVER to reset the default behavior.

See: "Reading Past the End of a Line" on page 867

See Also: FLOWOVER on page 860, MISSOVER on page 861, SCANOVER on page 863, and TRUNCOVER on page 863

Featured in: Example 2 on page 869

TRUNCOVER
overrides the default behavior of the INPUT statement when an input data record is shorter than the INPUT statement expects. By default, the INPUT statement automatically reads the next input data record. TRUNCOVER enables you to read variable-length records when some records are shorter than the INPUT statement expects. Variables without any values assigned are set to missing.

Tip: Use TRUNCOVER to assign the contents of the input buffer to a variable when the field is shorter than expected.

See: "Reading Past the End of a Line" on page 867

See Also: FLOWOVER on page 860, MISSOVER on page 861, SCANOVER on page 863, and STOPOVER on page 863

UNBUFFERED
tells SAS not to perform a buffered ("look ahead") read.

Alias: UNBUF

Interaction: When you use UNBUFFERED, SAS never sets the END= variable to 1.

Tip: When you read in-stream data with a DATALINES statement, UNBUFFERED is in effect.

INFILE=*variable*
names a character variable that references the contents of the current input buffer for this INFILE statement. You can use the variable in the same way as any other variable, even as the target of an assignment. The variable is automatically retained and initialized to blanks. Like automatic variables, the _INFILE_= variable is not written to the data set.

Restriction: *variable* cannot be a previously defined variable. Make sure that the _INFILE_= specification is the first occurence of this variable in the DATA step. Do not set or change the length of _INFILE_= variable with the LENGTH or ATTRIB statements. However, you can attach a format to this variable with the ATTRIB or FORMAT statement.

Interaction: The maximum length of this character variable is the logical record length for the specified INFILE statement. However, SAS does not open the file to know the LRECL= until prior to the execution phase. Therefore, the designated size for this variable during the compilation phase is 32,767.

Tip: Modification of this variable directly modifies the INFILE statement's current input buffer. Any PUT _INFILE_ (when this INFILE is current) that follows the buffer modification reflects the modified buffer contents. The _INFILE_= variable accesses only the current input buffer of the specified INFILE statement even if you use the N= option to specify multiple buffers.

Tip: To access the contents of the input buffer in another statement without using the _INFILE_= option, use the automatic variable _INFILE_.

Main Discussion: "Accessing the Contents of the Input Buffer" on page 865

Host Options

Operating Environment Information: For descriptions of operating environment-specific options in the INFILE statement, see the SAS documentation for your operating environment. △

DBMS-Specifications

DBMS-Specifications
enable you to read records from some DBMS files. You must license SAS/ACCESS software to be able to read from DBMS files. See the SAS/ACCESS documentation for the DBMS that you use.

Details

Operating Environment Information: The INFILE statement contains operating environment-specific material. See the SAS documentation for your operating environment before using this statement. △

How to Use the INFILE Statement Because the INFILE statement identifies the file to read, it must execute before the INPUT statement that reads the input data records. You can use the INFILE statement in conditional processing, such as an IF-THEN

statement, because it is executable. This allows you to control the source of the input data records.

Usually, you use an INFILE statement to read data from an external file. When data are read from the job stream, you must use a DATALINES statement. However, to take advantage of certain data-reading options that are available only in the INFILE statement, you can use an INFILE statement with the file-specification DATALINES and a DATALINES statement in the same DATA step.

When you use more than one INFILE statement for the same file-specification and you use options in each INFILE statement, the effect is additive. To avoid confusion, use all the options in the first INFILE statement for a given external file.

Reading Multiple Input Files You can read from multiple input files in a single iteration of the DATA step in one of two ways:

- to keep multiple files open and change which file is read, use multiple INFILE statements.
- To dynamically change the current input file within a single DATA step, use the FILEVAR= option in an INFILE statement. The FILEVAR= option enables you to read from one file, close it, and then open another. See Example 4 on page 870.

.

Updating External Files in Place You can use the INFILE statement in combination with the FILE statement to update records in an external file. Follow these steps:

1. Specify the INFILE statement before the FILE statement.
2. Specify the same fileref or physical filename in each statement.
3. Use options that are common to both the INFILE and FILE statements in the INFILE statement instead of the FILE statement. (Any such options that are used in the FILE statement are ignored.)

See Example 5 on page 871.

To update individual fields within a record instead of the entire record, see SHAREBUFFERS on page 863.

Accessing the Contents of the Input Buffer In addition to the _INFILE_= variable, you can use the automatic _INFILE_ variable to reference the contents of the current input buffer for the most recent execution of the INFILE statement. This character variable is automatically retained and initialized to blanks. Like other automatic variable, _INFILE_ is not written to the data set.

When you specify the _INFILE_= option in a INFILE statement then this variable is also indirectly referenced by the automatic _INFILE_ variable. If the automatic _INFILE_ variable is present and you omit _INFILE_= in a particular INFILE statement, then SAS creates an internal _INFILE_= variable for that INFILE statement. Otherwise, SAS does not create the _INFILE_= variable for a particular FILE.

During execution and at the point of reference, the maximum length of this character variable is the maximum length of the current _INFILE_= variable. However, because _INFILE_ merely references other variables whose lengths are not known until prior to the execution phase, the designated length is 32,767 during the compilation phase. For example, if you assign _INFILE_ to a new variable whose length is undefined, the default length of the new variable is 32,767. You can not use the LENGTH statement and the ATTRIB statement to set or override the length of _INFILE_. You can use the FORMAT statement and the ATTRIB statement to assign a format to _INFILE_.

Like other SAS variables, you can update the _INFILE_ variable in an assignment statement. You can also use a format with _INFILE_ in a PUT statement. For example

```
put _infile_ $hex100.;
```

outputs the contents of the input buffer using a hexadecimal format.

Any modification of the _INFILE_ directly modifies the current input buffer for the current INFILE statement. The execution of any PUT _INFILE_ statement that follows this buffer modification will reflect the contents of the modified buffer.

INFILE only accesses the contents of the current input buffer for a INFILE statement, even when you use the N= option to specify multiple buffers. You can access all the N= buffers, but you must use a INPUT statement with the # line pointer control to make the desired buffer the current input buffer.

Reading Delimited Data By default, the delimiter to read input data records with list input is a blank space. Both the DSD option and the DELIMITER= option affect how list input handles delimiters. The DELIMITER= option specifies that the INPUT statement use a character other than a blank as a delimiter for data values that are read with list input. When the DSD option is in effect, the INPUT statement uses a comma as the default delimiter.

To read a value as missing between two consecutive delimiters, use the DSD option. By default, the INPUT statement treats consecutive delimiters as a unit. When you use DSD, the INPUT statement treats consecutive delimiters separately. Therefore, a value that is missing between consecutive delimiters is read as a missing value. To change the delimiter from a comma to another value, use the DELIMITER= option.

For example, this DATA step program uses list input to read data that are separated with commas. The second data line contains a missing value. Because SAS allows consecutive delimiters with list input, the INPUT statement cannot detect the missing value.

```
data scores;
   infile datalines delimiter=',';
   input test1 test2 test3;
   datalines;
91,87,95
97,,92
,1,1
;
```

With the FLOWOVER option in effect, the data set SCORES contains two, not three, observations. The second observation is built incorrectly:

OBS	TEST1	TEST2	TEST3
1	91	87	95
2	97	92	1

To correct the problem, use the DSD option in the INFILE statement.

```
infile datalines dsd;
```

Now the INPUT statement detects the two consecutive delimiters and therefore assigns a missing value to variable TEST 2 in the second observation.

OBS	TEST1	TEST2	TEST3
1	91	87	95
2	97	.	92
3	1	1	1

The DSD option also enables list input to read a character value that contains a delimiter within a quoted string. For example, if data are separated with commas, DSD enables you to place the character string in quotation marks and read a comma as a valid character. SAS does not store the quotation marks as part of the character value. To retain the quotation marks as part of the value, use the tilde (~) format modifier in an INPUT statement. See Example 1 on page 868.

Reading Past the End of a Line By default, if the INPUT statement tries to read past the end of the current input data record, it moves the input pointer to column 1 of the next record to read the remaining values. This default behavior is specified by the FLOWOVER option. A message is written to the SAS log:

```
NOTE: SAS went to a new line when INPUT
@'CHARACTER_STRING' scanned past the end of a line.
```

The STOPOVER option treats this condition as an error and stops building the data set. The MISSOVER option sets the remaining INPUT statement variables to missing values. The SCANOVER option scans the input record until it finds the specified *character-string*. The FLOWOVER option restores the default behavior.

The TRUNCOVER option, like the MISSOVER option, overrides the default behavior of the INPUT statement. The MISSOVER option, however, causes the INPUT statement to set a value to missing if the statement is unable to read an entire field because the field length specified in the INPUT statement is too short. The TRUNCOVER option writes whatever characters are read to the appropriate variable so that you know what the input data record contained.

For example, an external file with variable-length records contains these records:

```
----+----1----+----2
1
22
333
4444
55555
```

The following DATA step reads these data to create a SAS data set. Only one of the input records is as long as the informatted length of the variable TESTNUM.

```
data numbers;
   infile 'external-file';
   input testnum 5.;
run;
```

This DATA step creates the three observations from the five input records because by default the FLOWOVER option is used to read the input records.

If you use the MISSOVER option in the INFILE statement, the DATA step creates five observations. However, all the values that were read from records that were too short are set to missing. Use the TRUNCOVER option in the INFILE statement to correct this problem:

```
infile 'external-file' truncover;
```

The DATA step now reads the same input records and creates five observations. See Table 6.5 on page 868 to compare the SAS data sets.

Table 6.5 The Value of TESTNUM Using Different INFILE Statement Options

OBS	FLOWOVER	MISSOVER	TRUNCOVER
1	22	.	1
2	4444	.	22
3	55555	.	333
4		.	4444
5		55555	55555

Comparisons

- The INFILE statement specifies the *input file* for any INPUT statements in the DATA step. The FILE statement specifies the *output file* for any PUT statements in the DATA step.
- An INFILE statement usually identifies data from an external file. A DATALINES statement indicates that data follow in the job stream. You can use the INFILE statement with the file specification DATALINES to take advantage of certain data-reading options that effect how the INPUT statement reads in-stream data.

Examples

Example 1: Changing How Delimiters are Treated By default, the INPUT statement uses a blank as the delimiter. This DATA step uses a comma as the delimiter:

```
data num;
   infile datalines dsd;
   input x y z;
   datalines;
,2,3
4,5,6
7,8,9
;
```

The argument DATALINES in the INFILE statement allows you to use an INFILE statement option to read in-stream data lines. The DSD option sets the comma as the default delimiter. Because a comma precedes the first value in the first dataline, a missing value is assigned to variable X in the first observation, and the value **2** is assigned to variable Y.

If the data uses multiple delimiters or a single delimiter other than a comma, simply specify the delimiter values with the DELIMITER= option. In this example, the characters a and b function as delimiters:

```
data nums;
   infile datalines dsd delimiter='ab';
   input X Y Z;
   datalines;
  1aa2ab3
  4b5bab6
```

```
7a8b9
;
```

The output that PROC PRINT generates shows the resulting NUMS data set. Values are missing for variables in the first and second observation because DSD causes list input to detect two consecutive delimiters. If you omit DSD, the characters a, b, aa, ab, ba, or bb function as the delimiter and no variables are assigned missing values.

Output 6.1 The NUMS Data Set

```
                          The SAS System                          1

                        OBS    X    Y    Z

                         1     1    .    2
                         2     4    5    .
                         3     7    8    9
```

This DATA step uses modified list input and the DSD option to read data that are separated by commas and that may contain commas as part of a character value:

```
data scores;
   infile datalines dsd;
   input Name : $9. Score
         Team  : $25. Div $;
   datalines;
Joseph,76,"Red Racers, Washington",AAA
Mitchel,82,"Blue Bunnies, Richmond",AAA
Sue Ellen,74,"Green Gazelles, Atlanta",AA
;
```

The output that PROC PRINT generates shows the resulting SCORES data set. The delimiter (comma) is stored as part of the value of TEAM while the quotation marks are not. The folowing output shows how to use the tilde (~) format modifier in an INPUT statement to retain the quotation marks in character data.

Output 6.2 Data Set SCORES

```
                          The SAS System                          1

 OBS    NAME          SCORE             TEAM                DIV

  1     Joseph          76     Red Racers, Washington      AAA
  2     Mitchel         82     Blue Bunnies, Richmond      AAA
  3     Sue Ellen       74     Green Gazelles, Atlanta     AA
```

Example 2: Handling Missing Values and Short Records with List Input This example demonstrates how to prevent missing values from causing problems when you read the data with list input. Some data lines in this example contain fewer than 5 temperature values. Use the MISSOVER option so that these values are set to missing.

```
data weather;
   infile datalines missover;
   input temp1-temp5;
```

```
   datalines;
97.9 98.1 98.3
98.6 99.2 99.1 98.5 97.5
96.2 97.3 98.3 97.6 96.5
;
```

SAS reads the three values on the first data line as the values of TEMP1, TEMP2, and TEMP3. The MISSOVER option causes SAS to set the values of TEMP4 and TEMP5 to missing for the first observation because no values for those variables are in the current input data record.

When you omit MISSOVER option or use FLOWOVER, SAS moves the input pointer to line 2 and reads values for TEMP4 and TEMP5. The next time the DATA step executes, SAS reads a new line which, in this case, is line 3. This message appears in the SAS log:

```
NOTE: SAS went to a new line when INPUT statement
      reached past the end of a line.
```

You can also use the STOPOVER option in the INFILE statement. This causes the DATA step to halt execution when an INPUT statement does not find enough values in a record of raw data:

```
 infile datalines stopover;
```

Because SAS does not find a TEMP4 value in the first data record, it sets _ERROR_ to 1, stops building the data set, and prints data line 1.

Example 3: Reading Files That Contain Variable-Length Records This example shows how to use LENGTH=, in combination with the $VARYING. informat, to read a file that contains variable-length records:

```
data a;
   infile file-specification length=linelen;
   input firstvar 1-10 @;  /* assign LINELEN   */
   varlen=linelen-10;      /* Calculate VARLEN */
   input @11 secondvar $varying500. varlen;
run;
```

The following occurs in this DATA step:

- The INFILE statement creates the variable LINELEN but does not assign it a value.
- When the first INPUT statement executes, SAS determines the line length of the record and assigns that value to the variable LINELEN. The single trailing @ holds the record in the input buffer for the next INPUT statement.
- The assignment statement uses the two known lengths (the length of FIRSTVAR and the length of the entire record) to determine the length of VARLEN.
- The second INPUT statement uses the VARLEN value with the informat $VARYING500. to read the variable SECONDVAR.

See the informat “$VARYING*w*.” on page 657 for more information.

Example 4: Reading from Multiple Input Files The following DATA step reads from two input files during each iteration of the DATA step. As SAS switches from one file to the next, each file remains open. The input pointer remains in place to begin reading from that location the next time an INPUT statement reads from that file.

```
data qtrtot(drop=jansale febsale marsale
                 aprsale maysale junsale);
```

```
      /* identify location of 1st file */
    infile file-specification-1;
      /* read values from 1st file     */
    input name $ jansale febsale marsale;
    qtr1tot=sum(jansale,febsale,marsale);

      /* identify location of 2nd file */
    infile file-specification-2;
      /* read values from 2nd file     */
    input @7 aprsale maysale junsale;
    qtr2tot=sum(aprsale,maysale,junsale);
run;
```

The DATA step terminates when SAS reaches an end-of-file on the shortest input file.

This DATA step uses FILEVAR= to read from a different file during each iteration of the DATA step:

```
data allsales;
   length fileloc myinfile $ 300;
   input fileloc $ ; /* read instream data       */

  /* The INFILE statement closes the current file
     and opens a new one if FILELOC changes value
     when INFILE executes                        */
   infile file-specification filevar=fileloc
          filename=myinfile end=done;

  /* DONE set to 1 when last input record read  */
   do while(not done);
  /* Read all input records from the currently  */
  /* opened input file, write to ALLSALES       */
     input name $ jansale febsale marsale;
     output;
   end;
   put 'Finished reading ' myinfile=;
   datalines;
external-file-1
external-file-2
external-file-3
;
```

The FILENAME= option assigns the name of the current input file to the variable MYINFILE. The LENGTH statement ensures that the FILENAME= variable and FILEVAR= variable have a length long enough to contain the value of the filename. The PUT statement prints the physical name of the currently open input file to the SAS log.

Example 5: Updating an External File This example shows how to use the INFILE statement with the SHAREBUFFERS option and the INPUT, FILE, and PUT statements to update an external file in place:

```
data _null_;
     /* The INFILE and FILE statements     */
     /* must specify the same file.       */
   infile file-specification-1 sharebuffers;
   file file-specification-1;
   input state $ 1-2 phone $ 5-16;
     /* Replace area code for NC exchanges */
```

```
   if state= 'NC' and substr(phone,5,3)='333' then
     phone='910-'||substr(phone,5,8);
   put phone 5-16;
run;
```

Example 6: Truncating Copied Records The LENGTH= option is useful when you copy the input file to another file with the PUT _INFILE_ statement. Use LENGTH= to truncate the copied records. For example, these statements truncate the last 20 columns from each input data record before the input data record is written to the output file:

```
data _null_;
   infile  file-specification-1 length=a;
   input;
   a=a-20;
   file file-specification-2;
   put _infile_;
run;
```

The START= option is also useful when you want to truncate what the PUT _INFILE_ statement copies. For example, if you do not want to copy the first 10 columns of each record, these statements copy from column 11 to the end of each record in the input buffer:

```
data _null_;
   infile file-specification start=s;
   input;
   s=11;
   file file-specification-2;
   put _infile_;
run;
```

Example 7: Listing the Pointer Location This DATA step assigns the value of the current pointer location in the input buffer to the variables LINEPT and COLUMNPT:

```
data _null_;
   infile datalines n=2 line=Linept col=Columnpt;
   input name $ 1-15 #2 @3 id;
   put linept= columnpt=;
   datalines;
J. Brooks
  40974
T. R. Ansen
  4032
;
```

These statements produce the following line for each execution of the DATA step because the input pointer is on the second line in the input buffer when the PUT statement executes:

```
Linept=2 Columnpt=9
Linept=2 Columnpt=8
```

See Also

Statements:

"FILENAME" on page 821
"INPUT" on page 876
"PUT" on page 962

INFORMAT

Associates informats with variables

Valid: in a DATA step or PROC step
Category: Information
Type: Declarative

Syntax

INFORMAT *variable(s)* <*informat*>
<DEFAULT=*default-informat*>;

Arguments

variable
names one or more variables to associate with an informat. You must specify at least one *variable*.

Tip: To disassociate an informat from a variable, use the variable's name in an INFORMAT statement without specifying an informat. Place the INFORMAT statement after the SET statement. See Example 3 on page 876.

informat
specifies the informat for reading the values of the variables that are listed in the INFORMAT statement.

Tip: Informats that are associated with variables by using an INFORMAT statement behave like informats that you specify with a colon (:) modifier in an INPUT statement. SAS reads the variables by using list input with an informat. For example, you can use the : modifier with an informat to read character values that are longer than eight bytes, or numeric values that contain nonstandard values. For details, see "INPUT, List" on page 897 .

See Also:

Featured in: Example 2 on page 875

DEFAULT= *default-informat*
specifies a temporary default informat for reading values of the variables that are listed in the INFORMAT statement. These default informats apply only to the current DATA step.

A DEFAULT= informat specification applies to

- variables that are not named in an INFORMAT or ATTRIB statement
- variables that are not permanently associated with an informat within a SAS data set
- variables that are not read with an explicit informat in the current DATA step.

Note: When you use the DEFAULT= option on the INFORMAT statement, SAS reads values as formatted input, regardless of the style of input that you specify on the INPUT statement. △

Default: If you omit DEFAULT=, SAS uses *w.d* as the default numeric informat and $*w*. as the default character informat.

Tip A DEFAULT= specification can occur anywhere in an INFORMAT statement. It can specify either a numeric default, a character default, or both.

Featured in: Example 1 on page 875

Details

The Basics An INFORMAT statement in a DATA step permanently associates an informat with a variable. You can specify standard SAS informats or user-written informats, previously defined in PROC FORMAT. A single INFORMAT statement can associate the same informat with several variables, or it can associate different informats with different variables. If a variable appears in multiple INFORMAT statements, SAS uses the informat that is assigned last.

CAUTION:
Because an INFORMAT statement defines the length of previously undefined character variables, you can truncate the values of character variables in a DATA step if an INFORMAT statement precedes a SET statement. △

How SAS Treats Variables when You Assign Informats with the INFORMAT Statement Informats that are associated with variables by using the INFORMAT statement behave like informats that are used with modified list input. SAS reads the variables by using the scanning feature of list input, but applies the informat. In modified list input, SAS

- does not use the value of *w* in an informat to specify column positions or input field widths in an external file
- uses the value of *w* in an informat to specify the length of previously undefined character variables
- ignores the value of *w* in numeric informats
- uses the value of *d* in an informat in the same way it usually does for numeric informats
- treats blanks that are embedded as input data as delimiters unless you change their status with a DELIMITER= option specification in an INFILE statement.

If you have coded the INPUT statement to use another style of input, such as formatted input or column input, that style of input is not used when you use the INFORMAT statement.

Comparisons

- Both the ATTRIB and INFORMAT statements can associate informats with variables, and both statements can change the informat that is associated with a variable. You can also use the INFORMAT statement in PROC DATASETS to change or remove the informat that is associated with a variable. The SAS windowing environment allows you to associate, change, or disassociate informats and variables in existing SAS data sets.
- SAS changes the descriptor information of the SAS data set that contains the variable. You can use an INFORMAT statement in some PROC steps, but the rules are different. See “The FORMAT Procedure” in *SAS Procedures Guide* for more information.

Examples

Example 1: Specifying Default Informats This example uses an INFORMAT statement to associate a default numeric informat and a default character informat:

```
data tstinfmt;
   informat default=3.1 default=$char4.;
   input x1-x10 name $;
   put x1-x10 name;
   datalines;
1112223334445556667778889991 00Johnny
;
```

The PUT statement produces this result :

```
----+----1----+----2----+----3----+----4----+----5----+
11.1 22.2 33.3 44.4 55.5 66.6 77.7 88.8 99.9 10 John
```

Example 2: Specifying Numeric and Character Informats This example associates a character informat and a numeric informat with SAS variables. Although the character variables do not fully occupy 15 column positions, the INPUT statement reads the data records correctly by using modified list input:

```
data name;
   informat FirstName LastName $15. n1 6.2 n2 7.3;
   input firstname lastname n1 n2;
   datalines;
Alexander Robinson 35 11
;

proc contents data=name;
run;

proc print data=name;
run;
```

The following output shows a partial listing from PROC CONTENTS, as well as the report PROC PRINT generates.

Output 6.3

```
                          The SAS System                          3

                       CONTENTS PROCEDURE

      -----Alphabetic List of Variables and Attributes-----

        #    Variable     Type    Len    Pos    Informat
        ------------------------------------------------
        1    FirstName    Char     15     16    $15.
        2    LastName     Char     15     31    $15.
        3    n1           Num       8      0    6.2
        4    n2           Num       8      8    7.3
```

```
                          The SAS System                          4

        OBS    FirstName    LastName     n1        n2

         1     Alexander    Robinson    0.35    0.011
```

Example 3: Removing an Informat This example disassociates an existing informat. The order of the INFORMAT and SET statements is important.

```
data rtest;
   set rtest;
   informat x;
run;
```

See Also

Statements:

"ATTRIB" on page 762
"INPUT" on page 876
"INPUT, List" on page 897

INPUT

Describes the arrangement of values in the input data record and assigns input values to the corresponding SAS variables

Valid: in a DATA step
Category: File-handling
Type: Executable

Syntax

INPUT <*specification(s)*><@|@@>;

Syntax Description

Without Arguments The INPUT statement with no arguments is called a *null INPUT statement*.

The null input statement

- brings an input data record into the input buffer without creating any SAS variables
- releases an input data record that is held by a trailing @ or a double trailing @.

Featured in: Example 2 on page 887

Arguments

specification
can include

variable
names a variable that is assigned input values.

(*variable-list*)
specifies a list of variables that are assigned input values.

Requirement: The (*variable-list*) is followed by an (*informat-list*).

See Also: "How to Group Variables and Informats" on page 895

$
indicates to store the variable value as a character value rather than as a numeric value.

Tip: If the variable is previously defined as character, $ is not required.

Featured in: Example 1 on page 887

pointer-control
moves the input pointer to a specified line or column in the input buffer.

See: "Column Pointer Controls" on page 878 and "Line Pointer Controls" on page 879

column-specifications
specifies the columns of the input record that contain the value to read.

See: "Column Input" on page 881

Featured in: Example 1 on page 887

format-modifier
allows modified list input or controls the amount of information that is reported in the SAS log when an error in an input value occurs.

Tip: Use modified list input to read data that cannot be read with simple list input.

See: "When to Use List Input" on page 898

See: "Format Modifiers for Error Reporting" on page 880

Featured in: Example 6 on page 889

informat.
specifies an informat to use to read the variable value.

Tip: You can use modified list input to read data with informats. This is useful when the data require informats but cannot be read with formatted input because the values are not aligned in columns.

See: "Formatted Input" on page 881 and "List Input" on page 881

Featured in: Example 2 on page 896

(*informat-list*)
specifies a list of informats to use to read the values for the preceding list of variables.

Restriction: The (*informat-list*) must follow the (*variable-list*).

See: "How to Group Variables and Informats" on page 895

@
holds an input record for the execution of the next INPUT statement within the same iteration of the DATA step. This line-hold specifier is called *trailing* @.

Restriction: The trailing @ must be the last item in the INPUT statement.

Tip: The trailing @ prevents the next INPUT statement from automatically releasing the current input record and reading the next record into the input buffer. It is useful when you need to read from a record multiple times.

See Also: "Using Line-Hold Specifiers" on page 883

Featured in: Example 3 on page 887

@@
holds the input record for the execution of the next INPUT statement across iterations of the DATA step. This line-hold specifier is called *double trailing* @.

Restriction: The double trailing @ must be the last item in the INPUT statement.

Tip: The double trailing @ is useful when each input line contains values for several observations.

See Also: "Using Line-Hold Specifiers" on page 883

Featured in: Example 4 on page 888

Column Pointer Controls

@*n*
moves the pointer to column *n*.

Range: a positive integer

Tip: If *n* is not an integer, SAS truncates the decimal value and uses only the integer value. If *n* is zero or negative, the pointer moves to column 1.

Example: @15 moves the pointer to column 15:

```
input @15 name $10.;
```

Featured in: Example 7 on page 890

@*numeric-variable*
moves the pointer to the column given by the value of *numeric-variable*.

Range: a positive integer

Tip: If *numeric-variable* is not an integer, SAS truncates the decimal value and only uses the integer value. If *numeric-variable* is zero or negative, the pointer moves to column 1.

Example: The value of the variable A moves the pointer to column 15:

```
a=15;
input @a name $10.;
```

Featured in: Example 5 on page 888

@(*expression*)
moves the pointer to the column that is given by the value of *expression*.

Restriction: *Expression* must result in a positive integer.

Tip: If the value of *expression* is not an integer, SAS truncates the decimal value and only uses the integer value. If it is zero or negative, the pointer moves to column 1.

Example: The result of the expression moves the pointer to column 15:

```
b=5;
input @(b*3) name $10.;
```

@'*character-string*'
locates the specified series of characters in the input record and moves the pointer to the first column after *character-string*.

@*character-variable*
locates the series of characters in the input record that is given by the value of *character-variable* and moves the pointer to the first column after that series of characters.

Example: The following statement reads in the WEEKDAY character variable. The second @1 moves the pointer to the beginning of the input line. The value for SALES is read from the next nonblank column after the value of WEEKDAY:

```
input @1 day 1. @5 weekday $10.
      @1 @weekday sales 8.2;
```

Featured in: Example 6 on page 889

@(*character-expression*)
locates the series of characters in the input record that is given by the value of *character-expression* and moves the pointer to the first column after the series.

Featured in: Example 6 on page 889

+*n*
moves the pointer *n* columns.

Range: a positive integer or zero

Tip: If *n* is not an integer, SAS truncates the decimal value and uses only the integer value. If the value is greater than the length of the input buffer, the pointer moves to column 1 of the next record.

Example: This statement moves the pointer to column 23, reads a value for LENGTH from columns 23 through 26, advances the pointer five columns, and reads a value for WIDTH from columns 32 through 35:

```
input @23 length 4. +5 width 4.;
```

Featured in: Example 7 on page 890

+*numeric-variable*
moves the pointer the number of columns that is given by the value of *numeric-variable*.

Range: a positive or negative integer or zero

Tip: If *numeric-variable* is not an integer, SAS truncates the decimal value and uses only the integer value. If *numeric-variable* is negative, the pointer moves backward. If the current column position becomes less than 1, the pointer moves to column 1. If the value is zero, the pointer does not move. If the value is greater than the length of the input buffer, the pointer moves to column 1 of the next record.

Featured in: Example 7 on page 890

+(*expression*)
moves the pointer the number of columns given by *expression*.

Range: *expression* must result in a positive or negative integer or zero.

Tip: If *expression* is not an integer, SAS truncates the decimal value and uses only the integer value. If *expression* is negative, the pointer moves backward. If the current column position becomes less than 1, the pointer moves to column 1. If the value is zero, the pointer does not move. If the value is greater than the length of the input buffer, the pointer moves to column 1 of the next record.

Line Pointer Controls

#*n*
moves the pointer to record *n*.

Range: a positive integer

Interaction: The N= option in the INFILE statement can affect the number of records the INPUT statement reads and the placement of the input pointer after each iteration of the DATA step. See the option N= on page 862.

Example: The #2 moves the pointer to the second record to read the value for ID from columns 3 and 4:

```
input name $10. #2 id 3-4;
```

#numeric-variable
moves the pointer to the record that is given by the value of *numeric-variable*.

Range: a positive integer

Tip: If the value of *numeric-variable* is not an integer, SAS truncates the decimal value and uses only the integer value.

#(*expression*)
moves the pointer to the record that is given by the value of *expression*.

Range: *expression* must result in a positive integer.

Tip: If the value of *expression* is not an integer, SAS truncates the decimal value and uses only the integer value.

/
advances the pointer to column 1 of the next input record.

Example: The values for NAME and AGE are read from the first input record before the pointer moves to the second record to read the value of ID from columns 3 and 4:

```
input name age / id 3-4;
```

Format Modifiers for Error Reporting

?
suppresses printing the invalid data note when SAS encounters invalid data values.

??
suppresses printing the messages and the input lines when SAS encounters invalid data values. The automatic variable _ERROR_ is not set to 1 for the invalid observation.

Details

When to Use INPUT Use the INPUT statement to read raw data from an external file or in-stream data. If your data are stored in an external file, you can specify the file in an INFILE statement. The INFILE statement must execute before the INPUT statement that reads the data records. If your data are in-stream, a DATALINES statement must precede the data lines in the job stream. If your data contain semicolons, use a DATALINES4 statement before the data lines. A DATA step that reads raw data can include multiple INPUT statements.

You can also use the INFILE statement to read in-stream data by specifying a filename of DATALINES on the INFILE statement before the INPUT statement. This allows you to use most of the options available on the INFILE statement with in-stream data.

To read data that are already stored in a SAS data set, use a SET statement. To read database or PC file-format data that are created by other software, use the SET statement after you access the data with the LIBNAME statement. See the SAS/ACCESS documentation for more information.

Input Styles

There are four ways to describe a record's values in the INPUT statement:

- column
- list (simple and modified)
- formatted
- named.

Each variable value is read by using one of these input styles. An INPUT statement may contain any or all of the available input styles, depending on the arrangement of data values in the input records. However, once named input is used in an INPUT statement, you cannot use another input style.

Column Input With *column input*, the column numbers follow the variable name in the INPUT statement. These numbers indicate where the variable values are found in the input data records:

```
input name $ 1-8 age 11-12;
```

This INPUT statement can read the following data records:

```
----+----1----+----2----+
Peterson  21
Morgan    17
```

Because NAME is a character variable, a $ appears between the variable name and column numbers. For more information, see "INPUT, Column" on page 890.

List Input With *list input*, the variable names are simply listed in the INPUT statement. A $ follows the name of each character variable:

```
input name $ age;
```

This INPUT statement can read data values that are separated by blanks or aligned in columns (with at least one blank between):

```
----+----1----+----2----+
Peterson  21
Morgan  17
```

For more information, see "INPUT, List" on page 897.

Formatted Input With *formatted input*, an informat follows the variable name in the INPUT statement. The informat gives the data type and the field width of an input value. Informats also allow you to read data that are stored in nonstandard form, such as packed decimal, or numbers that contain special characters such as commas.

```
input name $char8. +2 age 2.;
```

This INPUT statement reads these data records correctly:

```
----+----1----+----2----+
Peterson  21
Morgan    17
```

The pointer control of +2 moves the input pointer to the field that contains the value for the variable AGE. For more information, see "INPUT, Formatted" on page 893.

Named Input With *named input*, you specify the name of the variable followed by an equal sign. SAS looks for a variable name and an equal sign in the input record:

```
input name= $ age=;
```

This INPUT statement reads the following data records correctly:

```
----+----1----+----2----+
name=Peterson age=21
name=Morgan age=17
```

For more information, see “INPUT, Named” on page 902.

Multiple Styles in a Single INPUT Statement

An INPUT statement can contain any or all of the different input styles:

```
input idno name $18. team $ 25-30 startwght endwght;
```

This INPUT statement reads the following data records correctly:

```
----+----1----+----2----+----3----+----
023 David Shaw         red    189 165
049 Amelia Serrano     yellow 189 165
```

The value of IDNO, STARTWGHT, and ENDWGHT are read with list input, the value of NAME with formatted input, and the value of TEAM with column input.

Note: Once named input is used in an INPUT statement, you cannot change input styles. △

Pointer Controls

As SAS reads values from the input data records into the input buffer, it keeps track of its position with a pointer. The INPUT statement provides three ways to control the movement of the pointer:

column pointer controls
: reset the pointer's column position when the data values in the data records are read.

line pointer controls
: reset the pointer's line position when the data values in the data records are read.

line-hold specifiers
: hold an input record in the input buffer so that another INPUT statement can process it. By default, the INPUT statement releases the previous record and reads another record.

With column and line pointer controls, you can specify an absolute line number or column number to move the pointer or you can specify a column or line location relative to the current pointer position. Table 6.6 on page 882 lists the pointer controls that are available with the INPUT statement.

Table 6.6 Pointer Controls Available in the INPUT Statement

Pointer Controls	Relative	Absolute
column pointer controls	+*n*	@*n*
	+*numeric-variable*	@*numeric-variable*
	+(*expression*)	@(*expression*)
		@'*character-string*'

Pointer Controls	Relative	Absolute
		@character-variable
		@(*character-expression*)
line pointer controls	/	*#n*
		#numeric-variable
		#(*expression*)
line-hold specifiers	@	(not applicable)
	@@	(not applicable)

Note: Always specify pointer controls before the variable to which they apply. △

You can use the COLUMN= and LINE= options in the INFILE statement to determine the pointer's current column and line location.

Using Column and Line Pointer Controls Column pointer controls indicate the column in which an input value starts.

Use line pointer controls at the end of the INPUT statement to move to the next input record or to define the number of input records per observation. Line pointer controls specify which input record to read. To read multiple data records into the input buffer, use the N= option in the INFILE statement to specify the number of records. If you omit N=, you need to take special precautions. For more information, see .

Using Line-Hold Specifiers Line-hold specifiers keep the pointer on the current input record when

- a data record is read by more than one INPUT statement (trailing @)
- one input line has values for more than one observation (double trailing @).

Use a single trailing @ to allow the next INPUT statement to read from the same record. Use a double trailing @ to hold a record for the next INPUT statement across iterations of the DATA step.

Normally, each INPUT statement in a DATA step reads a new data record into the input buffer. When you use a trailing @, the following occurs:

- The pointer position does not change.
- No new record is read into the input buffer.
- The next INPUT statement for the same iteration of the DATA step continues to read the same record rather than a new one.

SAS releases a record held by a trailing @ when

- a null INPUT statement executes:

```
input;
```

- an INPUT statement without a trailing @ executes
- the next iteration of the DATA step begins.

Normally, when you use a double trailing @ (@@), the INPUT statement for the next iteration of the DATA step continues to read the same record. SAS releases the record that is held by a double trailing @

- immediately if the pointer moves past the end of the input record
- immediately if a null INPUT statement executes:

```
input;
```

- when the next iteration of the DATA step begins if an INPUT statement with a single trailing @ executes later in the DATA step:

```
input @;
```

Pointer Location After Reading Understanding the location of the input pointer after a value is read is important, especially if you combine input styles in a single INPUT statement. With column and formatted input, the pointer reads the columns that are indicated in the INPUT statement and stops in the next column. With list input, however, the pointer scans data records to locate data values and reads a blank to indicate that a value has ended. After reading a value with list input, the pointer stops in the second column after the value.

For example, you can read these data records with list, column, and formatted input:

```
----+----1----+----2----+----3
REGION1    49670
REGION2    97540
REGION3    86342
```

This INPUT statement uses list input to read the data records:

```
input region $ jansales;
```

After reading a value for REGION, the pointer stops in column 9.

```
----+----1----+----2----+----3
REGION1    49670
        ↑
```

These INPUT statements use column and formatted input to read the data records:

- column input

```
input region $ 1-7 jansales 12-16;
```

- formatted input

```
input region $7. +4 jansales 5.;
```

To read a value for the variable REGION, both INPUT statements instruct the pointer to read 7 columns and stop in column 8.

```
----+----1----+----2----+----3
REGION1    49670
       ↑
```

Reading More Than One Record per Observation

The highest number that follows the # pointer control in the INPUT statement determines how many input data records are read into the input buffer. Use the N= option in the INFILE statement to change the number of records. For example, in this statement, the highest value after the # is 3:

```
input @31 age 3. #3 id 3-4 #2 @6 name $20.;
```

Unless you use N= in the associated INFILE statement, the INPUT statement reads three input records each time the DATA step executes.

When each observation has multiple input records but values from the last record are not read, you must use a # pointer control in the INPUT statement or N= in the INFILE statement to specify the last input record. For example, if there are four records per observation, but only values from the first two input records are read, use this INPUT statement:

```
input name $ 1-10 #2 age 13-14 #4;
```

When you have advanced to the next record with the / pointer control, use the # pointer control in the INPUT statement or the N= option in the INFILE statement to set the number of records that are read into the input buffer. To move the pointer back to an earlier record, use a # pointer control. For example, this statement requires the #2 pointer control, unless the INFILE statement uses the N= option, to read two records:

```
input a / b #1 @52 c #2;
```

The INPUT statement assigns A a value from the first record. The pointer advances to the next input record to assign B a value. Then the pointer returns from the second record to column 1 of the first record and moves to column 52 to assign C a value. The #2 pointer control identifies two input records for each observation so that the pointer can return to the first record for the value of C.

If the number of input records per observation varies, use the N= option in the INFILE statement to give the maximum number of records per observation. For more information, see the N= option on page 862.

Reading Past the End of a Line When you use @ or + pointer controls with a value that moves the pointer to or past the end of the current record and the next value is to be read from the current column, SAS goes to column 1 of the next record to read it. It also writes this message to the SAS log:

```
NOTE: SAS went to a new line when INPUT statement
      reached past the end of a line.
```

You can alter the default behavior (the FLOWOVER option) in the INFILE statement.

Use the STOPOVER option in the INFILE statement to treat this condition as an error and to stop building the data set.

Use the MISSOVER option in the INFILE statement to set the remaining INPUT statement variables to missing values if the pointer reaches the end of a record.

Use the TRUNCOVER option in the INFILE statement to read column input or formatted input when the last variable that is read by the INPUT statement contains varying-length data.

Positioning the Pointer Before the Record When a column pointer control tries to move the pointer to a position before the beginning of the record, the pointer is positioned in column 1. For example, this INPUT statement specifies that the pointer is located in column –2 after the first value is read:

```
data test;
   input a @(a-3) b;
   datalines;
 2
;
```

Therefore, SAS moves the pointer to column 1 after the value of A is read. Both variables A and B contain the same value.

How Invalid Data are Handled

When SAS encounters an invalid character in an input value for the variable indicated, it

- sets the value of the variable that is being read to missing or the value that is specified with the INVALIDDATA= system option. For more information see “INVALIDDATA=” on page 1109.
- prints an invalid data note in the SAS log.

- prints the input line and column number that contains the invalid value in the SAS log. Unprintable characters appear in hexadecimal. To help determine column numbers, SAS prints a rule line above the input line.
- sets the automatic variable _ERROR_ to 1 for the current observation.

The format modifiers for error reporting control the amount of information that is printed in the SAS log. Both the ? and ?? modifier suppress the invalid data message. However, the ?? modifier also resets the automatic variable _ERROR_ to 0. For example, these two sets of statements are equivalent:

- `input x ?? 10-12;`
- ```
  input x ? 10-12;
  _error_=0;
  ```

In either case, SAS sets invalid values of X to missing values. For information on the causes of invalid data, see *SAS Language Reference: Concepts*.

End-of-File

End-of-file occurs when an INPUT statement reaches the end of the data. If a DATA step tries to read another record after it reaches an end-of-file then execution stops. If you want the DATA step to continue to execute, use the END= or EOF= option in the INFILE statement. Then you can write SAS program statements to detect the end-of-file, and to stop the execution of the INPUT statement but continue with the DATA step. For more information, see “INFILE” on page 857.

Arrays

The INPUT statement can use array references to read input data values. You can use an array reference in a pointer control if it is enclosed in parentheses. See Example 6 on page 889.

Use the array subscript asterisk (*) to input all elements of a previously defined explicit array. SAS allows single or multidimensional arrays. Enclose the subscript in braces, brackets, or parentheses. The form of this statement is

```
INPUT array-name{*};
```

You can use arrays with list, column, or formatted input. However, you cannot input values to an array that is defined with _TEMPORARY_ and that uses the asterisk subscript. For example, these statements create variables X1 through X100 and assign data values to the variables using the 2. informat:

```
array x{100};
input x{*} 2.;
```

Comparisons

- The INPUT statement reads raw data in external files or data lines that are entered in-stream (following the DATALINES statement) that need to be described to SAS. The SET statement reads a SAS data set, which already contains descriptive information about the data values.
- The INPUT statement reads data while the PUT statement writes data values and/or text strings to the SAS log or to an external file.
- The INPUT statement can read data from external files; the INFILE statement points to that file and has options that control how that file is read.
```

Examples

Example 1: Using Multiple Styles of Input in One INPUT Statement This example uses several input styles in a single INPUT statement:

```
data club1;
   input Idno Name $18.
         Team $ 25-30 Startwght Endwght;
   datalines;
023 David Shaw         red    189 165
049 Amelia Serrano     yellow 189 165
... more data lines ...
;
```

The values for ...	Are read with ...
Idno, Startwght, Endwght	list input
Name	formatted input
Team	column input

Example 2: Using a Null INPUT Statement This example uses an INPUT statement with no arguments. The DATA step copies records from the input file to the output file without creating any SAS variables:

```
data _null_;
   infile file-specification-1;
   file file-specification-2;
   input;
   put _infile_;
run;
```

Example 3: Holding a Record in the Input Buffer This example reads a file that contains two kinds of input data records and creates a SAS data set from these records. One type of data record contains information about a particular college course. The second type of record contains information about the students enrolled in the course. You need two INPUT statements to read the two records and to assign the values to different variables that use different formats. Records that contain class information have a C in column 1; records that contain student information have an S in column 1, as shown here:

```
----+----1----+----2----+
C HIST101 Watson
S Williams 0459
S Flores   5423
C MATH202 Sen
S Lee      7085
```

To know which INPUT statement to use, check each record as it is read. Use an INPUT statement that reads only the variable that tells whether the record contains class or student.

```
data schedule(drop=type);
   infile file-specification;
   retain Course Professor;
   input type $ 1 @;
```

```
   if type='C' then
      input course $ professor $;
   else if type='S' then
      do;
        input Name $10. Id;
        output schedule;
      end;
run;


proc print;
run;
```

The first INPUT statement reads the TYPE value from column 1 of every line. Because this INPUT statement ends with a trailing @, the next INPUT statement in the DATA step reads the same line. The IF-THEN statements that follow check whether the record is a class or student line before another INPUT statement reads the rest of the line. The INPUT statements without a trailing @ release the held line. The RETAIN statement saves the values about the particular college course. The DATA step writes an observation to the SCHEDULE data set after a student record is read.

The following output that PROC PRINT generates shows the resulting data set SCHEDULE.

Output 6.4 Data Set Schedule

```
                        The SAS System                          1

       OBS    Course     Professor    Name          Id

        1     HIST101     Watson      Williams      459
        2     HIST101     Watson      Flores       5423
        3     MATH202     Sen         Lee          7085
```

Example 4: Holding a Record Across Iterations of the DATA Step This example shows how to create multiple observations for each input data record. Each record contains several NAME and AGE values. The DATA step reads a NAME value and an AGE value, outputs an observation, then reads another set of NAME and AGE values to output, and so on until all the input values in the record are processed.

```
data test;
   input name $ age @@;
   datalines;
John 13 Monica 12 Sue 15 Stephen 10
Marc 22 Lily 17
;
```

The INPUT statement uses the double trailing @ to control the input pointer across iterations of the DATA step. The SAS data set contains six observations.

Example 5: Positioning the Pointer with a Numeric Variable This example uses a numeric variable to position the pointer. A raw data file contains records with the employment figures for several offices of a multinational company. The input data records are

```
----+----1----+----2----+----3----+
8      New York    1 USA 14
```

```
5   Cary           1 USA 2274
3 Chicago          1 USA 37
22 Tokyo           5 ASIA 80
5   Vancouver      2 CANADA 6
9       Milano     4 EUROPE 123
```

The first column has the column position for the office location. The next numeric column is the region category. The geographic region occurs before the number of employees in that office.

You determine the office location by combining the *@numeric-variable* pointer control with a trailing @. To read the records, use two INPUT statements. The first INPUT statement obtains the value for the *@ numeric-variable* pointer control. The second INPUT statement uses this value to determine the column that the pointer moves to.

```
data office (drop=x);
   infile file-specification;
   input x @;
   if 1<=x<=10 then
      input @x City $9.;
   else do;
      put 'Invalid input at line ' _n_;
      delete;
   end;
run;
```

The DATA step writes only five observations to the OFFICE data set. The fourth input data record is invalid because the value of X is greater than 10. Therefore, the second INPUT statement does not execute. Instead, the PUT statement writes a message to the SAS log and the DELETE statement stops processing the observation.

Example 6: Positioning the Pointer with a Character Variable This example uses character variables to position the pointer. The OFFICE data set, created in Example 5 on page 888, contains a character variable CITY whose values are the office locations. Suppose you discover that you need to read additional values from the raw data file. By using another DATA step, you can combine the *@character-variable* pointer control with a trailing @ and the *@character-expression* pointer control to locate the values.

If the observations in OFFICE are still in the order of the original input data records, you can use this DATA step:

```
data office2;
   set office;
   infile file-specification;
   array region {5} $ _temporary_
         ('USA' 'CANADA' 'SA' 'EUROPE' 'ASIA');
   input @city Location : 2. @;
   input @(trim(region{location})) Population : 4.;
run;
```

The ARRAY statement assigns initial values to the temporary array elements. These elements correspond to the geographic regions of the office locations. The first INPUT statement uses an *@character-variable* pointer control. Each record is scanned for the series of characters in the value of CITY for that observation. Then the value of LOCATION is read from the next nonblank column. LOCATION is a numeric category for the geographic region of an office. The second INPUT statement uses an array reference in the *@character-expression* pointer control to determine the location POPULATION in the input records. The expression also uses the TRIM function to

trim trailing blanks from the character value. This way an exact match is found between the character string in the input data and the value of the array element.

The following output that PROC PRINT generates shows the resulting data set OFFICE2.

Output 6.5 Data Set Office2

```
                          The SAS System                          1

          OBS    City          Location    Population

            1    New York           1            14
            2    Cary               1          2274
            3    Chicago            1            37
            4    Vancouver          2             6
            5    Milano             4           123
```

Example 7: Moving the Pointer Backward This example shows several ways to move the pointer backward.

- This INPUT statement uses the @ pointer control to read a value for BOOK starting at column 26. Then the pointer moves back to column 1 on the same line to read a value for COMPANY:

```
input @26 book $ @1 company;
```

- These INPUT statements use +*numeric-variable* or +(*expression*) to move the pointer backward one column. These two sets of statements are equivalent.
  -
```
m=-1;
input x 1-10 +m y 2.;
```
  -
```
input x 1-10 +(-1) y 2.;
```

See Also

Statements:

“ARRAY” on page 755
“INPUT, Column” on page 890
“INPUT, Formatted” on page 893
“INPUT, List” on page 897
“INPUT, Named” on page 902

INPUT, Column

Reads input values from specified columns and assigns them to the corresponding SAS variables

Valid: in a DATA step
Category: File-handling
Type: Executable

Syntax

INPUT *variable* <$> *start-column* <— *end-column*>

<*.decimals*> <@ | @@>;

Arguments

variable
names a variable that is assigned input values.

$
indicates that the variable has character values rather than numeric values.

Tip: If the variable is previously defined as character, $ is not required.

start-column
specifies the first column of the input record that contains the value to read.

— *end-column*
specifies the last column of the input record that contains the value to read.

Tip: If the variable value occupies only one column, omit *end-column*.

Example: Because *end-column* is omitted, the values for the character variable GENDER occupy only column 6:

```
input name $ 1-10 pulse 11-13 waist 14-15
      gender $ 16;
```

.decimals
specifies the number of digits to the right of the decimal if the input value does not contain an explicit decimal point.

Tip: An explicit decimal point in the input value overrides a decimal specification in the INPUT statement.

Example: This INPUT statement reads the input data for a numeric variable using two decimal places:

Input Data	Statement	Results
----+---1		
2314	**input number 1-5 .2;**	**23.14**
2		**.02**
400		**4.00**
-140		**-1.40**
12.234		**12.234** *
12.2		**12.2** *

* The decimal specification in the INPUT statement is overridden by the input data value.

@
holds the input record for the execution of the next INPUT statement within the same iteration of the DATA step. This line-hold specifier is called *trailing @*.

Restriction: The trailing @ must be the last item in the INPUT statement.

Tip: The trailing @ prevents the next INPUT statement from automatically releasing the current input record and reading the next record into the input buffer. It is useful when you need to read from a record multiple times.

See: .

@@
holds the input record for the execution of the next INPUT statement across iterations of the DATA step. This line-hold specifier is called *double trailing @*.

Restriction: The double trailing @ must be the last item in the INPUT statement.

Tip: The double trailing @ is useful when each input line contains values for several observations.

See: "Using Line-Hold Specifiers" on page 883.

Details

When to Use Column Input With column input, the column numbers that contain the value follow a variable name in the INPUT statement. To read with column input, data values must be in

- the same columns in all the input data records
- standard numeric form or character form.*

Useful features of column input are that

- Character values can contain embedded blanks.
- Character values can be from 1 to 32,767 characters long.
- Input values can be read in any order, regardless of their position in the record.
- Values or parts of values can be read multiple times. For example, this INPUT statement reads an ID value in columns 10 through 15 and then reads a GROUP value from column 13:

```
input id 10-15 group 13;
```

- Both leading and trailing blanks within the field are ignored. Therefore, if numeric values contain blanks that represent zeros or if you want to retain leading and trailing blanks in character values, read the value with an informat. See "INPUT, Formatted" on page 893.

Missing Values Missing data do not require a place-holder. The INPUT statement interprets a blank field as missing and reads other values correctly. If a numeric or character field contains a single period, the variable value is set to missing.

Reading Data Lines SAS always pads the data records that follow the DATALINES statement (in-stream data) to a fixed length in multiples of 80. The CARDIMAGE system option determines whether to read or to truncate data past the 80th column.

Reading Variable-Length Records By default, SAS uses the FLOWOVER option to read varying-length data records. If the record contains fewer values than expected, the INPUT statement reads the values from the next data record. To read varying-length data. you may need to use the TRUNCOVER option in the INFILE statement. The TRUNCOVER option is more efficient than the PAD option which pads the records to a fixed length. For more information, see "Reading Past the End of a Line" on page 867.

Examples

This DATA step demonstrates how to read input data records with column input:

---

* See *SAS Language Reference: Concepts* for the definition of standard and nonstandard data values.

```
data scores;
   input name $ 1-18 score1 25-27 score2 30-32
         score3 35-37;
   datalines;
Joseph                  11   32   76
Mitchel                 13   29   82
Sue Ellen               14   27   74
;
```

See Also

Statement:

"INPUT" on page 876

INPUT, Formatted

Reads input values with specified informats and assigns them to the corresponding SAS variables

Valid in a DATA step

Category: File-handling

Type: Executable

Syntax

INPUT <*pointer-control*> *variable informat.* <@ | @@>;

INPUT<*pointer-control*> (*variable-list*) (*informat-list*)
<@ | @@>;

INPUT <*pointer-control*> (*variable-list*) (<*n**> *informat.*)
<@ | @@>;

Arguments

pointer-control
moves the input pointer to a specified line or column in the input buffer.

See: "Column Pointer Controls" on page 878 and "Line Pointer Controls" on page 879

variable
names a variable that is assigned input values.

Requirement: The (*variable-list*) is followed by an (*informat-list*).

Featured in: Example 1 on page 896

(*variable-list*)
specifies a list of variables that are assigned input values.

See: "How to Group Variables and Informats" on page 895

Featured in: Example 2 on page 896

informat.
specifies a SAS informat to use to read the variable values.

Tip: Decimal points in the actual input values override decimal specifications in a numeric informat.

See Also: Chapter 5, " Informats," on page 629

Featured in: Example 1 on page 896

(*informat-list*)
specifies a list of informats to use to read the values for the preceding list of variables In the INPUT statement, (*informat-list*) can include

informat.
specifies an informat to use to read the variable values.

pointer-control
specifies one of these pointer controls to use to position a value: @, #, /, or +.

*n**
specifies to repeat *n* times the next informat in an informat list.

Example: This statement uses the 7.2 informat to read GRADES1, GRADES2, and GRADES3 and the 5.2 informat to read GRADES4 and GRADES5:

```
input (grades1-grades5)(3*7.2, 2*5.2);
```

Restriction: The (*informat-list*) must follow the (*variable-list*).

See: "How to Group Variables and Informats" on page 895

Featured in: Example 2 on page 896

@
holds an input record for the execution of the next INPUT statement within the same iteration of the DATA step. This line-hold specifier is called *trailing @*.

Restriction: The trailing @ must be the last item in the INPUT statement.

Tip: The trailing @ prevents the next INPUT statement from automatically releasing the current input record and reading the next record into the input buffer. It is useful when you need to read from a record multiple times.

See: "Using Line-Hold Specifiers" on page 883

@@
holds an input record for the execution of the next INPUT statement across iterations of the DATA step. This line-hold specifier is called *double trailing @*.

Restriction: The double trailing @ must be the last item in the INPUT statement.

Tip: The double trailing @ is useful when each input line contains values for several observations.

See: "Using Line-Hold Specifiers" on page 883

Details

When to Use Formatted Input With formatted input, an informat follows a variable name and defines how SAS reads the values of this variable. An informat gives the data type and the field width of an input value. Informats also read data that are stored in nonstandard form, such as packed decimal, or numbers that contain special characters such as commas.* See "Definition" on page 631 for descriptions of SAS informats.

* See *SAS Language Reference: Concepts* for information on standard and nonstandard data values.

Simple formatted input requires that the variables be in the same order as their corresponding values in the input data. You can use pointer controls to read variables in any order. For more information, see "INPUT" on page 876.

Missing Values Generally, SAS represents missing values in formatted input with a single period for a numeric value and with blanks for a character value. The informat that you use with formatted input determines how SAS interprets a blank. For example, $CHAR.*w* reads the blanks as part of the value, whereas BZ.*w* converts a blank to zero.

Reading Variable-Length Records By default, SAS uses the FLOWOVER option to read varying-length data records. If the record contains fewer values than expected, the INPUT statement reads the values from the next data record. To read varying-length data. you may need to use the TRUNCOVER option in the INFILE statement. For more information, see "Reading Past the End of a Line" on page 867.

How to Group Variables and Informats When the input values are arranged in a pattern, you can group the informat list. A grouped informat list consists of two lists:

- the names of the variables to read enclosed in parentheses
- the corresponding informats separated by either blanks or commas and enclosed in parentheses.

Informat lists can make an INPUT statement shorter because the informat list is recycled until all variables are read and the numbered variable names can be used in abbreviated form. This avoids listing the individual variables.

For example, if the values for the five variables SCORE1 through SCORE5 are stored as four columns per value without intervening blanks, this INPUT statement reads the values:

```
input (score1-score5) (4. 4. 4. 4. 4.);
```

However, if you specify more variables than informats, the INPUT statement reuses the informat list to read the remaining variables. A shorter version of the previous statement is

```
input (score1-score5) (4.);
```

You can use as many informat lists as necessary in an INPUT statement, but do not nest the informat lists. After all the values in the variable list are read, the INPUT statement ignores any directions that remain in the informat list. In this example the value of X is read with the 2. informat :

```
data test;
   input (x y z) (2.,+1);
   datalines;
2 24 36
0 20 30
   ;
```

The +1 column pointer control moves the pointer forward one column after X is read. The value of Y is read with the 2. informat. Again, the +1 column pointer moves the pointer forward one column. Then, the value of Z is read with the 2. informat. For the third iteration, the INPUT statement ignores the +1 pointer control.

The *n** modifier in an informat list specifies to repeat the next informat *n* times. For example,

```
input (name score1-score5) ($10. 5*4.);
```

How to Store Informats The informats that you specify in the INPUT statement are not stored with the SAS data set. Informats that you specify with the INFORMAT or

ATTRIB statement are permanently stored. Therefore, you can read a data value with a permanently stored informat in a later DATA step without having to specify the informat or use PROC FSEDIT to enter data in the correct format.

Comparisons

When a variable is read with formatted input, the pointer movement is similar to that of column input. The pointer moves the length that the informat specifies and stops at the next column. To read data with informats that are not aligned in columns, use *modified list input*. This allows you to take advantage of the scanning feature in list input. See "When to Use List Input" on page 898.

Examples

Example 1: Formatted Input with Pointer Controls This INPUT statement uses informats and pointer controls:

```
data sales;
   infile file-specification;
   input item $10. +5 jan comma5. +5 feb comma5.
         +5 mar comma5.;
run;
```

It can read these input data records:

```
----+----1----+----2----+----3----+----4
trucks          1,382     2,789     3,556
vans            1,265     2,543     3,987
sedans          2,391     3,011     3,658
```

The value for ITEM is read from the first 10 columns in a record. The pointer stops in column 11. The trailing blanks are discarded and the value of ITEM is written to the program data vector. Next, the pointer moves five columns to the right before the INPUT statement uses the COMMA5. informat to read the value of JAN. This informat uses five as the field width to read numeric values that contain a comma. Once again, the pointer moves five columns to the right before the INPUT statement uses the COMMA5. informat to read the values of FEB and MAR.

Example 2: Using Informat Lists This INPUT statement uses the character informat $10. to read the values of the variable NAME and uses the numeric informat 4. to read the values of the five variables SCORE1 through SCORE5:

```
data scores;
   input (name score1-score5) ($10. 5*4.);
   datalines;
Whittaker 121 114 137 156 142
Smythe    111 97  122 143 127
;
```

See Also

Statements:

"INPUT" on page 876

INPUT, List

Scans the input data record for input values and assigns them to the corresponding SAS variables

Valid: in a DATA step

Category: File-handling

Type: Executable

Syntax

INPUT <*pointer-control*> *variable* <$> <&> <@ | @@>;

INPUT <*pointer-control*> *variable* <:|&|~>
<*informat.*> <@ | @@>;

Arguments

pointer-control
moves the input pointer to a specified line or column in the input buffer.

See: "Column Pointer Controls" on page 878 and "Line Pointer Controls" on page 879

Featured in: Example 2 on page 901

variable
names a variable that is assigned input values.

$
indicates to store a variable value as a character value rather than as a numeric value.

Tip: If the variable is previously defined as character, $ is not required.

Featured in: Example 1 on page 900

&
indicates that a character value may have one or more single embedded blanks. This format modifier reads the value from the next nonblank column until the pointer reaches two consecutive blanks, the defined length of the variable, or the end of the input line, whichever comes first.

Restriction: The & modifier must follow the variable name and $ sign that it affects.

Tip: If you specify an informat after the & modifier, the terminating condition for the format modifier remains two blanks.

See: "Modified List Input" on page 899

Featured in: Example 2 on page 901

:
allows you to specify an informat that the INPUT statement uses to read the variable value. This format modifier reads the value from the next nonblank column until the pointer reaches the next blank column, the defined length of the variable, or the end of the data line, whichever comes first.

Tip: If the length of the variable is not previously defined, its value is read and stored with the informat length.

Tip: The pointer continues to read until the next blank column is reached. However, if the field is longer than the formatted length, the value is truncated to the length of variable.

See: "Modified List Input" on page 899

Featured in: Example 3 on page 901 and Example 5 on page 901

~
indicates to treat single quotation marks, double quotation marks, and delimiters in character values in a special way. This format modifier reads delimiters within quoted character values as characters instead of as delimiters and retains the quotation marks when the value is written to a variable.

Restriction: You must use the DSD option in an INFILE statement. Otherwise, the INPUT statement ignores this option.

See: "Modified List Input" on page 899

Featured in: Example 5 on page 901

informat.
specifies an informat to use to read the variable values.

Tip: Decimal points in the actual input values always override decimal specifications in a numeric informat.

See Also: "Definition" on page 631

Featured in: Example 3 on page 901 and Example 5 on page 901

@
holds an input record for the execution of the next INPUT statement within the same iteration of the DATA step. This line-hold specifier is called *trailing @*.

Restriction: The trailing @ must be the last item in the INPUT statement.

Tip: The trailing @ prevents the next INPUT statement from automatically releasing the current input record and reading the next record into the input buffer. It is useful when you need to read from a record multiple times.

See: "Using Line-Hold Specifiers" on page 883

@@
holds an input record for the execution of the next INPUT statement across iterations of the DATA step. This line-hold specifier is called *double trailing @*.

Restriction: The double trailing @ must be the last item in the INPUT statement.

Tip: The double trailing @ is useful when each input line contains values for several observations.

See: "Using Line-Hold Specifiers" on page 883

Details

When to Use List Input List input requires that you specify the variable names in the INPUT statement in the same order that the fields appear in the input data records. SAS scans the data line to locate the next value but ignores additional intervening blanks. List input does not require that the data are located in specific columns.

However, you must separate each value from the next by at least one blank unless the delimiter between values is changed. By default, the delimiter for data values is one blank space or the end of the input record. List input will not skip over any data values to read subsequent values, but it can ignore all values after a given point in the data record. However, pointer controls allow you to change the order that the data values are read.

There are two types of list input:

- simple list input
- modified list input.

Modified list input makes the INPUT statement more versatile because you can use a format modifier to overcome several of the restrictions of simple list input. See "Modified List Input" on page 899.

Simple List Input Simple list input places several restrictions on the type of data that the INPUT statement can read:

- By default, at least one blank must separate the input values. Use the DELIMITER= option or the DSD option in the INFILE statement to specify a delimiter other than a blank.
- Represent each missing value with a period, not a blank, or two adjacent delimiters.
- Character input values cannot be longer than 8 bytes unless the variable is given a longer length in an earlier LENGTH, ATTRIB, or INFORMAT statement.
- Character values cannot contain embedded blanks unless you change the delimiter.
- Data must be in standard numeric or character format.*

Modified List Input List input is more versatile when you use format modifiers. The format modifiers are as follows:

Format Modifier	Purpose
&	reads character values that contain embedded blanks.
:	reads data values that need the additional instructions that informats can provide but that are not aligned in columns. **
~	reads delimiters within quoted character values as characters and retains the quotation marks.

** Use formatted input and pointer controls to quickly read data values aligned in columns.

For example, use the : modifier with an informat to read character values that are longer than 8 bytes or numeric values that contain nonstandard values.

Because list input interprets a blank as a delimiter, use modified list input to read values that contain blanks. The & modifier reads character values that contain single embedded blanks. However, the data values must be separated by two or more blanks. To read values that contain leading, trailing, or embedded blanks with list input, use the DELIMITER= option in the INFILE statement to specify another character as the delimiter. See Example 5 on page 901. If your input data use blanks as delimiters and they contain leading, trailing, or embedded blanks, you may need to use either column input or formatted input. If quotation marks surround the delimited values, you can use list input with the DSD option in the INFILE statement.

* See *SAS Language Reference: Concepts* for the information on standard and nonstandard data values.

Comparisons

How Modified List Input and Formatted Input Differ *Modified list input* has a scanning feature that can use informats to read data which are not aligned in columns. *Formatted input* causes the pointer to move like that of column input to read a variable value. The pointer moves the length that is specified in the informat and stops at the next column.

This DATA step uses modified list input to read the first data value and formatted input to read the second:

```
data jansales;
   input item : $10. amount comma5.;
datalines;
trucks 1,382
vans 1,235
sedans 2,391
;
```

The value of ITEM is read with modified list input. The INPUT statement stops reading when the pointer finds a blank space. The pointer then moves to the second column after the end of the field, which is the correct position to read the AMOUNT value with formatted input.

Formatted input, on the other hand, continues to read the entire width of the field. This INPUT statement uses formatted input to read both data values:

```
input item $10. +1 amount comma5.;
```

To read these data correctly with formatted input, the second data value must occur after the 0^{th} column of the first value, as shown here:

```
----+----1----+----2
trucks    1,382
vans      1,235
sedans    2,391
```

Also, after the value of ITEM is read with formatted input, you must use the pointer control +1 to move the pointer to the column where the value AMOUNT begins.

When Data Contain Quotation Marks When you use the DSD option in an INFILE statement, which sets the delimiter to a comma, the INPUT statement removes quotation marks before a value is written to a variable. When you also use the tilde (~) modifier in an INPUT statement, the INPUT statement maintains quotation marks as part of the value.

Examples

Example 1: Reading Unaligned Data with Simple List Input The INPUT statement in this DATA step uses simple list input to read the input data records:

```
data scores;
   input name $ score1 score2 score3 team $;
   datalines;
Joe 11 32 76 red
Mitchel 13 29 82 blue
Susan 14 27 74 green
;
```

The next INPUT statement reads only the first four fields in the previous data lines, which demonstrates that you are not required to read all the fields in the record:

```
input name $ score1 score2 score3;
```

Example 2: Reading Character Data That Contain Embedded Blanks The INPUT statement in this DATA step uses the & format modifier with list input to read character values that contain embedded blanks.

```
data list;
   infile file-specification;
   input name $ & score;
run;
```

It can read these input data records:

```
----+----1----+----2----+----3----+
Joseph   11 Joergensen  red
Mitchel  13 Mc Allister  blue
Su Ellen  14 Fischer-Simon  green
```

The & modifier follows the variable it affects in the INPUT statement. Because this format modifier follows NAME, at least two blanks must separate the NAME field from the SCORE field in the input data records.

You can also specify an informat with a format modifier, as shown here:

```
    input name $ & +3 lastname & $15. team $;
```

In addition, this INPUT statement reads the same data to demonstrate that you are not required to read all the values in an input record. The +3 column pointer control moves the pointer past the score value in order to read the value for LASTNAME and TEAM.

Example 3: Reading Unaligned Data with Informats This DATA step uses modified list input to read data values with an informat:

```
data jansales;
   input item : $10. amount;
   datalines;
trucks 1382
vans 1235
sedans 2391
;
```

The $10. informat allows a character variable of up to ten characters to be read.

Example 4: Reading Delimited Data with Simple List Input This DATA step uses the DELIMITER= option in the INFILE statement to read data values that are separated by commas, instead of blanks, with simple list input:

```
data scores2;
   infile datalines delimiter=',';
   input name $ score1-score3 team $;
   datalines;
Joe,11,32,76,red
Mitchel,13,29,82,blue
Susan,14,27,74,green
;
```

Example 5: Reading Delimited Data with Modified List Input This DATA step uses the DSD option in an INFILE statement and the tilde (~) format modifier in an INPUT statement to retain the quotation marks in character data and to read a character in a quoted string as a character instead of as a delimiter.

```
data scores;
   infile datalines dsd;
   input Name : $9. Score1-Score3
         Team ~ $25. Div $;
   datalines;
Joseph,11,32,76,"Red Racers, Washington",AAA
Mitchel,13,29,82,"Blue Bunnies, Richmond",AAA
Sue Ellen,14,27,74,"Green Gazelles, Atlanta",AA
;
```

The output that PROC PRINT generates shows the resulting SCORES data set. The values for TEAM contain the quotation marks.

Output 6.6 SCORES Data Set

```
                          The SAS System                          1

OBS Name      Score1 Score2 Score3           Team             Div

 1  Joseph       11     32     76   "Red Racers, Washington"  AAA
 2  Mitchel      13     29     82   "Blue Bunnies, Richmond"  AAA
 3  Sue Ellen    14     27     74   "Green Gazelles, Atlanta" AA
```

See Also

Statements:

"INFILE" on page 857
"INPUT" on page 876
"INPUT, Formatted" on page 893

INPUT, Named

Reads data values that appear after a variable name that is followed by an equal sign and assigns them to corresponding SAS variables

Valid: in a DATA step
Category: File-handling
Type: Executable

Syntax

INPUT <*pointer-control*> *variable*= <$> <@ | @@>;

INPUT <*pointer-control*> *variable*= *informat.* <@ | @@>;

INPUT *variable*= <$> *start-column* <— *end-column*> <*.decimals*> <@ | @@>;

Arguments

pointer-control
moves the input pointer to a specified line or column in the input buffer.

See: "Column Pointer Controls" on page 878 and "Line Pointer Controls" on page 879

variable=
names a variable whose value is read by the INPUT statement. In the input data record, the field has the form

```
variable=value
```

Featured in: Example 1 on page 904

$
indicates to store a variable value as a character value rather than as a numeric value.

Tip: If the variable is previously defined as character, $ is not required.

Featured in: Example 1 on page 904

informat.
specifies an informat that indicates the data type of the input values, but not how the values are read.

Tip: Use the INFORMAT statement to associate an informat with a variable.

See: Chapter 5, " Informats," on page 629

Featured in: Example 1 on page 904

start-column
specifies the column that the INPUT statement uses to begin scanning in the input data records for the variable. The variable name does not have to begin here.

— *end-column*
determines the default length of the variable.

@
holds an input record for the execution of the next INPUT statement within the same iteration of the DATA step. This line-hold specifier is called *trailing @*.

Restriction: The trailing @ must be the last item in the INPUT statement.

Tip: The trailing @ prevents the next INPUT statement from automatically releasing the current input record and reading the next record into the input buffer. It is useful when you need to read from a record multiple times.

See: "Using Line-Hold Specifiers" on page 883

@@
holds an input record for the execution of the next INPUT statement across iterations of the DATA step. This line-hold specifier is called *double trailing @*.

Restriction: The double trailing @ must be the last item in the INPUT statement.

Tip: The double trailing @ is useful when each input line contains values for several observations.

See: "Using Line-Hold Specifiers" on page 883

Details

When to Use Named Input Named input reads the input data records that contain a variable name followed by an equal sign and a value for the variable. The INPUT statement reads the input data record at the current location of the input pointer. If the input data records contain data values at the start of the record that the INPUT

statement cannot read with named input, use another input style to read them. However, once the INPUT statement starts to read named input, SAS expects that all the remaining values are in this form. See Example 1 on page 904.

You do not have to specify the variables in the INPUT statement in the same order that they occur in the data records. Also, you do not have to specify a variable for each field in the record. However, if you do not specify a variable in the INPUT statement that another statement uses (for example, ATTRIB, FORMAT, INFORMAT, LENGTH statement) and it occurs in the input data record, the INPUT statement automatically reads the value. SAS writes a note to the log that the variable is uninitialized.

When you do not specify a variable for all the named input data values, SAS sets _ERROR_ to 1 and writes a note to the log. For example,

```
data list;
   input name=$ age=;
   datalines;
name=John age=34  gender=M
;
```

The note written to the log states that GENDER is not defined and _ERROR_ is set to 1.

Restrictions

- Once you start to read with named input, you cannot switch to another input style or use pointer controls. All the remaining values in the input data record must be in the form *variable=value*. SAS treats the values that are not in named input form as invalid data.
- If named input values continue after the end of the current input line, use a slash (/) at the end of the input line. This tells SAS to move the pointer to the next line and to continue to read with named input. For example,

  ```
  input name=$ age=;
  ```

  can read this input data record:

  ```
  name=John /
     age=34
  ```

- If you use named input to read character values that contain embedded blanks, put two blanks before and after the data value, as you would with list input. See Example 2 on page 905.
- You cannot reference an array with an asterisk or an expression subscript.

Examples

Example 1: Using Named Input with Another Input Style This DATA step uses list input and named input to read input data records:

```
options yearcutoff= 1920;

data list;
   input id name=$20. gender=$;
   informat dob ddmmyy8.;
   datalines;
4798  gender=m name=COLIN age=23 dob=16/02/75
2653 name=MICHELE age=46 gender=f
;
```

The INPUT statement uses list input to read the first variable, ID. The remaining variables NAME, GENDER, and DOB are read with named input. These variables are

not read in order. The $20. informat with NAME= prevents the INPUT statement from truncating the character value to a length of eight. The INPUT statement reads the DOB= field because the INFORMAT statement refers to this variable. It skips the AGE= field altogether. SAS writes notes to the log that DOB is uninitialized, AGE is not defined, and _ERROR_ is set to 1.

Example 2: Reading Character Variables with Embedded Blanks This DATA step reads character variables that contain embedded blanks with named input:

```
data list2;
   informat header $30. name $15.;
   input header= name=;
   datalines;
header=  age=60 AND UP  name=PHILIP
;
```

Two spaces precede and follow the value of the variable HEADER, which is **AGE=60 AND UP**. The field also contains an equal sign.

See Also

Statement:

"INPUT" on page 876

KEEP

Includes variables in output SAS data sets

Valid: With LENGTH= in a DATA step
Category: Information
Type: Declarative

Syntax

KEEP *variable-list*;

Arguments

variable-list
specifies the names of the variables to write to the output data set.

Tip: List the variables in any form that SAS allows.

Details

The KEEP statement causes a DATA step to write only the variables that you specify to one or more SAS data sets. The KEEP statement applies to all SAS data sets that are created within the same DATA step and can appear anywhere in the step. If no KEEP or DROP statement appears, all data sets that are created in the DATA step contain all variables.

Note: Do not use both the KEEP and DROP statements within the same DATA step. △

Comparisons

- The KEEP *statement* cannot be used in SAS PROC steps. The KEEP= *data set option* can.
- The KEEP *statement* applies to all output data sets that are named in the DATA statement. To write different variables to different data sets, you must use the KEEP= *data set option*.
- The DROP statement is a parallel statement that specifies variables to omit from the output data set.
- The KEEP and DROP statements select variables to include in or exclude from output data sets. The subsetting IF statement selects observations.
- Do not confuse the KEEP statement with the RETAIN statement. The RETAIN statement causes SAS to hold the value of a variable from one iteration of the DATA step to the next iteration. The KEEP statement does not affect the value of variables but only specifies which variables to include in any output data sets.

Examples

- These examples show the correct syntax for listing variables in the KEEP statement:
  - `keep name address city state zip phone;`
  - `keep rep1-rep5;`
- This example uses the KEEP statement to include only the variables NAME and AVG in the output data set. The variables SCORE1 through SCORE20, from which AVG is calculated, are not written to the data set AVERAGE.

```
data average;
   keep name avg;
   infile file-specification;
   input name $ score1-score20;
   avg=mean(of score1-score20);
run;
```

See Also

Data Set Option:

"KEEP=" on page 27

Statements:

"DROP" on page 796
"IF, Subsetting" on page 847
"RETAIN" on page 999

LABEL

Assigns descriptive labels to variables

Valid: in a DATA step
Category: Information
Type: Declarative

Syntax

LABEL *variable-1='label-1' . . . <variable-n='label-n'>*;

LABEL *variable-1=' ' . . . <variable-n=' '>*;

Arguments

variable
names the variable that you want to label.

Tip: Optionally, you can specify additional pairs of labels and variables.

'label'
specifies a label of up to 256 characters, including blanks.

Tip: Optionally, you can specify additional pairs of labels and variables.

Tip: For more information about including quotation marks as part of the label, see Character Constants in *SAS Language Reference: Concepts*.

Restriction: You must enclose the label in either single or double quotation marks.

' '
removes a label from a variable. Enclose a single blank space in quotation marks to remove an existing label.

Details

Using a LABEL statement in a DATA step permanently associates labels with variables by affecting the descriptor information of the SAS data set that contains the variables. You can associate any number of variables with labels in a single LABEL statement.

You can use a LABEL statement in a PROC step, but the rules are different. See the *SAS Procedures Guide* for more information.

Comparisons

Both the ATTRIB and LABEL statements can associate labels with variables and change a label that is associated with a variable.

Examples

Example 1: Specifying Labels Here are several LABEL statements:

```
label compound='Type of Drug';

label date="Today's Date ";

label n='Mark''s Experiment Number';

label score1="Grade on April 1 Test"
      score2="Grade on May 1 Test";
```

Example 2: Removing a Label This example removes an existing label:

```
data rtest;
   set rtest;
   label x=' ';
run;
```

See Also

Statement:
"ATTRIB" on page 762

Labels, Statement

Identifies a statement that is referred to by another statement

Valid: in a DATA step
Category: Control
Type: Declarative

Syntax

label: *statement*;

Arguments

label
specifies any SAS name, which is followed by a colon (:). You must specify the *label* argument.

statement
specifies any executable statement, including a null statement (;). You must specify the *statement* argument.

Restriction: No two statements in a DATA step can have the same label.

Restriction: If a statement in a DATA step is labeled, it should be referenced by a statement or option in the same step.

Tip: A null statement can have a label:

```
ABC:;
```

Details

The statement label identifies the destination of either a GO TO statement, a LINK statement, the HEADER= option in a FILE statement, or the EOF= option in an INFILE statement.

Comparisons

The LABEL statement assigns a descriptive label to a variable. A statement label identifies a statement or group of statements that are referred to in the same DATA step by another statement, such as a GO TO statement.

Examples

In this example, if Stock=0, the GO TO statement causes SAS to jump to the statement that is labeled reorder. When Stock is not 0, execution continues to the RETURN statement and then returns to the beginning of the DATA step for the next observation.

```
data Inventory Order;
   input Item $ Stock @;
      /* go to label reorder: */
   if Stock=0 then go to reorder;
   output Inventory;
   return;
      /* destination of GO TO statement */
   reorder: input Supplier $;
   put 'ORDER ITEM ' Item 'FROM ' Supplier;
   output Order;
   datalines;
milk  0 A
bread 3 B
;
```

See Also

Statements:

"GO TO" on page 845

"LINK" on page 923

Statement Options:

HEADER= option in the FILE statement on page 806

EOF= option in the INFILE statement on page 859

LEAVE

Stops processing the current loop and resumes with the next statement in sequence

Valid: in a DATA step

Category: Control

Type: Executable

Syntax

LEAVE;

Without Arguments

The LEAVE statement stops the processing of the current DO loop or SELECT group and continues DATA step processing with the next statement following the DO loop or SELECT group.

Details

You can use the LEAVE statement to exit a DO loop or SELECT group prematurely based on a condition.

Comparisons

- The LEAVE statement causes processing of the current loop to end. The CONTINUE statement stops the processing of the current iteration of a loop and resumes with the next iteration.
- You can use the LEAVE statement in a DO loop or in a SELECT group. You can use the CONTINUE statement only in a DO loop.

Examples

This DATA step demonstrates using the LEAVE statement to stop the processing of a DO loop under a given condition. In this example, the IF/THEN statement checks the value of BONUS. When the value of BONUS reaches 500, the maximum amount allowed, the LEAVE statement stops the processing of the DO loop.

```
data week;
   input name $ idno start_yr status $ dept $;
   bonus=0;
   do year= start_yr to 1991;
      if bonus ge 500 then leave;
      bonus+50;
   end;
   datalines;
Jones 9011 1990 PT PUB
Thomas 876 1976 PT HR
Barnes 7899 1991 FT TECH
Harrell 1250 1975 FT HR
Richards 1002 1990 FT DEV
Kelly 85 1981 PT PUB
Stone 091 1990 PT MAIT
;
```

LENGTH

Specifies the number of bytes for storing variables

Valid: in a DATA step

Category: Information

Type: Declarative

Syntax

LENGTH *variable-specification(s)*
<DEFAULT=*n*>;

Arguments

variable-specification
is a required argument and has the form *variable(s)* < $ >*length* where

variable
names one or more variables that are to be assigned a length. This includes any variables in the DATA step, including those dropped from the output data set.

Restriction: Array references are not allowed.

Tip: If the variable is character, the length applies to the program data vector and the output data set. If the variable is numeric, the length applies only to the output data set.

$
indicates that the preceding variables are character variables.

Default: SAS assumes that the variables are numeric.

length
specifies a numeric constant that is the number of bytes used for storing variable values.

Range: For numeric variables, 2 to 8 or 3 to 8, depending on your operating environment. For character variables, 1 to 32767 under all operating environments.

DEFAULT=*n*
changes the default number of bytes that SAS uses to store the values of any newly created numeric variables.

Default: 8

Range: 2 to 8 or 3 to 8, depending on your operating environment.

CAUTION:
Avoid shortening numeric variables that contain fractions. The precision of a numeric variable is closely tied to its length, especially when the variable contains fractional values. You can safely shorten variables that contain integers according to the rules that are given in the SAS documentation for your operating environment, but shortening variables that contain fractions may eliminate important precision.
△

Details

In general, the length of a variable depends on

- whether the variable is numeric or character
- how the variable was created
- whether a LENGTH or ATTRIB statement is present.

Subject to the rules for assigning lengths, lengths that are assigned with the LENGTH statement can be changed in the ATTRIB statement and vice versa. See “SAS Variables” in *SAS Language Reference: Concepts* for information on assigning lengths to variables.

Operating Environment Information: Valid variable lengths depend on your operating environment. For details, see the SAS documentation for your operating environment. △

Comparisons

The ATTRIB statement can assign the length as well as other attributes of variables.

Examples

This example uses a LENGTH statement to set the length of the character variable NAME to 25. It also changes the default number of bytes that SAS uses to store the values of newly created numeric variables from 8 to 4. The TRIM function removes trailing blanks from LASTNAME before it is concatenated with a comma (,) , a blank space, and the value of FIRSTNAME. If you omit the LENGTH statement, SAS sets the length of NAME to 32.

```
data testlength;
   informat FirstName LastName $15. n1 6.2;
   input firstname lastname n1 n2;
   length name $25 default=4;
   name=trim(lastname)||', '||firstname;
   datalines;
Alexander Robinson 35 11
;

proc contents data=testlength;
run;

proc print data=testlength;
run;
```

The following output shows a partial listing from PROC CONTENTS, as well as the report that PROC PRINT generates.

Output 6.7

```
                          The SAS System                          3

                       CONTENTS PROCEDURE

      -----Alphabetic List of Variables and Attributes-----

        #    Variable     Type    Len    Pos    Informat
        ----------------------------------------------
        1    FirstName    Char     15      8    $15.
        2    LastName     Char     15     23    $15.
        3    n1           Num       4      0    6.2
        4    n2           Num       4      4
        5    name         Char     25     38
```

```
                          The SAS System                          4

OBS    FirstName   LastName      n1      n2          name

 1     Alexander   Robinson   0.35000    11   Robinson, Alexander
```

See Also

Statement:

"ATTRIB" on page 762

For information on the use of the LENGTH statement in PROC steps, see *SAS Procedures Guide*

LIBNAME

Associates or disassociates a SAS data library with a libref (a shortcut name); clears one or all librefs; lists the characteristics of a SAS data library; concatenates SAS data libraries; implicitly concatenates SAS catalogs.

Valid: Anywhere

Category: Data Access

See Also: LIBNAME, SAS/ACCESS

Syntax

❶**LIBNAME** *libref* <*engine*> '*SAS-data-library*'
< *options* > <*engine/host-options*>;

❷**LIBNAME** *libref* CLEAR | _ALL_ CLEAR;

❸**LIBNAME** *libref* LIST | _ALL_ LIST;

❹❺**LIBNAME** *libref* <*engine*> (*library–specification–1* <. . . *library-specification-n*>)
< *options* > ;

libref
is a shortcut name or a "nickname" for the aggregate storage location where your SAS files are stored. It is any SAS name when you are assigning a new libref. When you are disassociating a libref from a SAS data library or when listing attributes, specify a libref that was previously assigned.

Tip: The association between a libref and a SAS data library lasts only for the duration of the SAS session or until you change it or discontinue it with another LIBNAME statement.

'SAS-data-library'
must be the physical name for the SAS data library. The physical name is the name that is recognized by the operating environment. Enclose the physical name in single or double quotation marks.

Operating Environment Information: For details on specifying the physical names of files, see the SAS documentation for your operating environment. △

In some operating environments, you can use an operating environment command, the LIBNAME statement, or the LIBNAME function to associate the libref with the SAS data library. In others, you must use an operating environment command. See the SAS documentation for your operating environment for more information.

library-specification
is two or more SAS data libraries, specified by physical names, previously-assigned librefs, or a combination of the two. Separate each specification with either a blank or a comma and enclose the entire list in parentheses.

'SAS-data-library'
the physical name of a SAS data library, enclosed in quotation marks.

libref
is the name of a previously-assigned libref.

Restriction: When concatenating libraries, you cannot specify options that are specific to an engine or an operating environment.

Featured in: Example 2 on page 918

See Also: "Rules for Library Concatenation" on page 917

engine
is an engine name.

Tip: Generally, SAS automatically determines the appropriate engine to use for accessing the files in the library. If you want to create a new library with an engine other than the default engine, you can override the automatic selection.

See: For a list of valid engines, see the SAS documentation for your operating environment. For background information about engines, see *SAS Language Reference: Concepts*.

CLEAR
disassociates one or more currently assigned librefs.

Tip: Specify *libref* to disassociate a single libref. Specify _ALL_ to disassociate all currently assigned librefs.

ALL
specifies that the CLEAR or LIST argument applies to all currently-assigned librefs.

LIST
writes the attributes of one or more SAS data libraries to the SAS log.

Tip: Specify *libref* to list the attributes of a single SAS data library. Specify _ALL_ to list the attributes of all SAS data libraries that have librefs in your current session.

Options

ACCESS=READONLY | TEMP

READONLY — assigns a read-only attribute to an entire SAS data library. SAS will not allow you to open a data set in the library in order to update information or write new information.

TEMP — indicates that the SAS data library be treated as a scratch library. That is, the system will not consume CPU cycles to ensure that the files in a TEMP library do not become corrupted.

Tip: Use ACCESS=TEMP to save resources only when the data are recoverable.

Operating Environment Information: Some operating environments support LIBNAME statement options that have similar functions to the ACCESS= option. See the SAS documentation for your operating environment. △

OUTREP=*default-format*

specifies the default format for the SAS data library that is specified. New data sets that are stored in this SAS data library are written in this format. Existing data sets that are written to the library are given the new format. Values for OUTREP= are

MVS
CMS
VSE
VAX_VMS
ALPHA_VMS
ALPHA_OSF
SOLARIS
HP_UX
RS_6000_AIX
WINDOWS
OS2
MAC

REPEMPTY=YES | NO

controls replacement of like-named temporary or permanent SAS data sets when the new one is empty.

YES — specifies that a new empty data set with a given name replaces an existing data set with the same name. This is the default.

Interaction: When REPEMPTY=YES and REPLACE=NO, then the data set is not replaced.

NO — specifies that a new empty data set with a given name does not replace an existing data set with the same name.

Tip: Use REPEMPTY=NO to prevent the following syntax error from replacing the existing data set MYLIB.B with the new empty data set MYLIB.B that is created by mistake:

```
libname libref SAS-data-library REPEMPTY=NO;
data mylib.a set mylib.b;
```

Tip: For both the convenience of replacing existing data sets with new ones that contain data and the protection of not overwriting existing data sets with new empty ones that are created by accident, set REPLACE=YES and REPEMPTY=NO.

Comparison: For an individual data set, the REPEMPTY= data set option overrides the setting of the REPEMPTY= option in the LIBNAME statement.

See Also: "REPEMPTY=" on page 36

TRANTAB=*translation-table*
specifies the translation table to use for character conversions. When any data set in a format foreign to the host is written to the specified SAS data library, SAS uses the specified translation table.

Interaction: A translation table that is specified with the TRANTAB= data set option for a specific data set overrides the translation table that is specified with the TRANTAB= option in the LIBNAME statement for an entire library.

Operating Environment Information: For a list of valid specifications, see the SAS documentation for your operating environment. △

Engine-Host-Options

engine-host-options
are one or more options listed in the general form *keyword=value*.

Operating Environment Information: For a list of valid specifications, see the SAS documentation for your operating environment. △

Restriction: When concatenating libraries, you cannot specify options that are specific to an engine or an operating environment.

Details

❶ Associating a Libref with a SAS Data Library

The association between a libref and a SAS data library lasts only for the duration of the SAS session or until you change it or discontinue it with another LIBNAME statement. The simplest form of the LIBNAME statement specifies only a libref and the physical name of a SAS data library:

LIBNAME *libref 'SAS-data-library'*;

See Example 1 on page 918.

An engine specification is usually not necessary. If the situation is ambiguous, SAS uses the setting of the ENGINE= system option to determine the default engine. If all data sets in the library are associated with a single engine, SAS uses that engine as the default. In either situation, you can override the default by specifying another engine with the ENGINE= option:

LIBNAME *libref engine 'SAS-data-library'*
<options > <engine/host-options>;

Operating Environment Information: Using the LIBNAME statement requires host-specific information. See the SAS documentation for your operating environment before using this statement. △

❷Disassociating a libref from a SAS Data Library To disassociate a libref from a SAS data library, use a LIBNAME statement, specifying the libref and the CLEAR option. You can clear a single, specified libref or all current librefs.

LIBNAME *libref* CLEAR | _ALL_ CLEAR;

❸Writing SAS Data Library Attributes to the SAS Log Use a LIBNAME statement to write the attributes of one or more SAS data libraries to the SAS log. Specify *libref* to list the attributes of one SAS data library; use _ALL_ to list the attributes of all SAS data libraries that have been assigned librefs in your current SAS session.

LIBNAME *libref* LIST | _ALL_ LIST;

❹Concatenating SAS Data Libraries When you logically concatenate two or more SAS data libraries, you can reference them all with one libref. You can specify a library with its physical pathname or its previously assigned libref.

LIBNAME *libref* <*engine*> (*library–specification–1* <. . . *library-specification-n*>) < *options* > ;

In the same LIBNAME statement you can use any combination of specifications: librefs, physical pathnames, or a combination of librefs and pathnames. See Example 2 on page 918.

❺Implicitly Concatenating SAS Catalogs When you logically concatenate two or more SAS data libraries, you also implicitly concatenate the SAS catalogs that have the same name. For example, if three SAS data libraries each contain a catalog named CATALOG1, then when you concatenate them, you implicitly create a catalog concatenation for the catalogs that have the same name. See Example 3 on page 919.

LIBNAME *libref* <*engine*> (*library–specification–1* <. . . *library-specification-n*>) < *options* > ;

Rules for Library Concatenation After you create a library concatenation, you can specify the libref in any context that accepts a simple (nonconcatenated) libref. These rules determine how SAS files (that is, members of SAS libraries) are located among the concatenated libraries:

1. When a SAS file is opened for input or update, the concatenated libraries are searched and the first occurrence of the specified file is used.
2. When a SAS file is opened for output, it is created in the first library that is listed in the concatenation.

   Note: A new SAS file is created in the first library even if there is a file with the same name in another part of the concatenation. △
3. When you delete or rename a SAS file, only the first occurrence of the file is affected.
4. Any time a list of SAS files is displayed, only one occurrence of a file name is shown.

   Note: Even if the name occurs multiple times in the concatenation, only the first occurrence is shown. △
5. A SAS file that is logically connected to another file (such as an index to a data set) is listed only if the parent file resides in that same library. For example, if library ONE contains A.DATA, and library TWO contains A.DATA and A.INDEX, only A.DATA from library ONE is listed. (See rule 4.)

6 If any library in the concatenation is sequential, then all of the libraries are treated as sequential.

7 The attributes of the first library that is specified determine the attributes of the concatenation. For example, if the first SAS data library that is listed is "read only", then the entire concatenated library is "read only".

8 If you specify any options or engines, they apply only to the libraries that you specified with the complete physical name, not to any library that you specified with a libref.

9 If you alter a libref after it has been assigned in a concatenation, it will not affect the concatenation.

Comparisons

- Use the LIBNAME statement to reference a SAS data library. Use the FILENAME statement to reference an external file. Use the LIBNAME, SAS/ACCESS statement to access DBMS tables.
- Use the CATNAME statement to *explicitly* concatenate SAS catalogs. Use the LIBNAME statement to *implicitly* concatenate SAS catalogs. The CATNAME statement allows you to specify the names of the catalogs that you want to concatenate. The LIBNAME statement concatenates all like-named catalogs in the specified SAS data libraries.

Examples

Example 1: Assigning and Using a Libref This example assigns the libref SALES to an aggregate storage location that is specified in quotation marks as a physical pathname. The DATA step creates SALES.QUARTER1 and stores it in that location. The PROC PRINT step references it by its two-level name, SALES.QUARTER1.

```
libname sales 'SAS-data-library';

data sales.quarter1;
   infile 'your-input-file';
   input salesrep $20. +6 jansales febsales
         marsales;
run;

proc print data=sales.quarter1;
run;
```

Example 2: Logically Concatenating SAS Data Libraries

- This example concatenates three SAS data libraries by specifying the physical filename of each:

```
libname allmine ('path-1' 'path-2'
                 'path-3');
```

- This example assigns librefs to two SAS data libraries, one that contains Version 6 SAS files and one that contains Version 7 SAS files. This technique is useful for updating your files and applications from Version 6 to Version 7, while allowing you to have convenient access to both sets of files:

```
libname v6 'path-to--v6--library';
libname v7 'path-to-v7--library';
```

```
libname allmine (v6 v7);
```

- This example shows that you can specify both librefs and physical filenames in the same concatenation specification:

```
libname allmine (v6 v7 'some-path');
```

Example 3: Implicitly Concatenating SAS Catalogs This example concatenates three SAS data libraries by specifying the physical filename of each and assigns the libref ALLMINE to the concatenated libraries:

```
libname allmine ('path-1' 'path-2'
                 'path-3');
```

If each library contains a SAS catalog named MYCAT, then using ALLMINE.MYCAT as a libref.catref provides access to the catalog entries that are stored in all three catalogs named MYCAT. To logically concatenate SAS catalogs with different names, see "CATNAME" on page 769.

Example 4: Storing Data Sets with One-Level Names Permanently If you want the convenience of specifying only a one-level name for permanent, not temporary, SAS files, use the USER= system option. This example stores data set QUARTER1 permanently without using a LIBNAME statement first to assign a libref to a storage location:

```
options user='SAS-data-library';

data quarter1;
   infile 'your-input-file;
   input salesrep $20. +6 jansales febsales
         marsales;
run;

proc print data=quarter1;
run;
```

See Also

Statements:

"CATNAME" on page 769 for a discussion of *explicitly* concatenating SAS catalogs

"FILENAME" on page 821

"LIBNAME, SAS/ACCESS" on page 919

System Option:

"USER=" on page 1172

LIBNAME, SAS/ACCESS

Associates a SAS libref with a database management system (DBMS) database, schema, server, or group of tables or views

Valid: Anywhere

Category: Data Access

Required: You must license SAS/ACCESS software in order to use the LIBNAME statement to access data that are stored in a DBMS file.

Syntax

❶**LIBNAME** *libref SAS/ACCESS-engine-name*
<SAS/ACCESS-engine-connection-options>
<SAS/ACCESS-engine-LIBNAME-options>;

❷**LIBNAME** *libref* CLEAR | _ALL_ CLEAR;

❸**LIBNAME** *libref* LIST | _ALL_ LIST;

❹**LIBNAME** *libref <SAS/ACCESS-engine>* (*library–specification–1 <. . . library-specification-n>*)
< options > ;

Arguments

libref
is a shortcut or a "nickname" for the DBMS database, schema, or server where your tables and views are stored. It is any SAS name when you are assigning a new libref. When you are disassociating a currently-assigned libref or when you are listing attributes with the LIBNAME statement, specify a libref that was previously assigned with a LIBNAME statement.

Tip: The association between a libref and a DBMS database, schema, or server lasts only for the duration of the SAS session or until you change it or discontinue it with another LIBNAME statement. You may change a libref as often as you want.

SAS/ACCESS-engine-name
is a SAS/ACCESS engine name for your DBMS, such as ORACLE or DB2. DBMS engines may be implemented differently in different operating environments. See *SAS/ACCESS Software for Relational Databases: Reference*.

Requirement: To access data from a DBMS table, you must specify *SAS/ACCESS-engine-name*.

CLEAR
disassociates one or more currently assigned librefs.

Tip: Specify *libref* to disassociate a single libref. Specify _ALL_ to disassociate all currently assigned librefs.

ALL
specifies that the CLEAR or LIST argument applies to all currently-assigned librefs.

LIST
writes the attributes of one or more SAS/ACCESS libraries or SAS data libraries to the SAS log.

Tip: Specify *libref* to list the attributes of a single SAS/ACCESS library or SAS data library. Specify _ALL_ to list the attributes of all libraries that have librefs in your current session.

SAS/ACCESS-Engine-Connection-Options

SAS/ACCESS-engine-connection-options
are options that you specify in order to connect to a particular database; these options are different for each database. For example, to connect to a database through ODBC, you specify your user name, password, data source, and other options. Enclose the SAS/ACCESS-engine-connection-options in quotation marks if they contain characters that are not allowed in SAS names.

See: the SAS/ACCESS documentation for documentation on these options. Support for many of these options is DBMS-specific.

SAS/ACCESS-Engine-LIBNAME-Options

SAS/ACCESS-engine-LIBNAME-options
specify actions that apply to the processing of the DBMS's tables. For example, SPOOL= specifies whether SAS creates a utility spool file during read transactions that read data more than once.

Interaction: Some SAS/ACCESS-engine-LIBNAME options have an equivalent data set option. For an individual table, you can override the value that is specified for the library in a LIBNAME statement by using the corresponding data set option after the table name in a DATA or PROC step.

See: the SAS/ACCESS documentation for documentation on these options. Support for many of these options is DBMS-specific.

Details

❶Using Data from a DBMS

If you have a license for SAS/ACCESS software, you can use a LIBNAME statement to read from and write to a DBMS table or view, as though it were in a SAS data set. The LIBNAME statement associates a libref with a SAS/ACCESS engine in order to access tables or views in a DBMS. The SAS/ACCESS engine enables you to connect to a particular DBMS and, therefore, to specify a DBMS table or view name in a two-level SAS name.

For example, consider this PROC step:

```
proc print data=mylib.employees_q2;
run;
```

MYLIB is a SAS libref that points to a particular DBMS, and EMPLOYEES_Q2 is a DBMS table name. When you specify MYLIB.EMPLOYEES_Q2 in a DATA step or PROC step, you dynamically access the DBMS table. The SAS System now supports reading, updating, and creating DBMS tables. See the SAS/ACCESS documentation for more information.

❷Disassociating a LIbref from a SAS Data Library

To disassociate a libref from a SAS/ACCESS library or a SAS data library, use a LIBNAME statement, specifying the libref and the CLEAR option. You can clear a single specified libref or all current librefs.

LIBNAME *libref* CLEAR | _ALL_ CLEAR;

❸Writing SAS Data Library Attributes to the SAS Log

Use a LIBNAME statement to write the attributes of one or more SAS/ACCESS libraries or SAS data libraries to the SAS log. Specify *libref* to list the attributes of a single SAS/ACCESS library or SAS data library. Specify _ALL_ to list the attributes of all libraries that have librefs in your current session

LIBNAME *libref* LIST | _ALL_ LIST;

❹Concatenating SAS Data Libraries When you logically concatenate two or more SAS data libraries, you can reference them all with one libref. You can specify a library with its physical pathname or its previously assigned libref.

LIBNAME *libref* <*SAS/ACCESS-engine*> (*library–specification–1* <. . . *library-specification-n*>)
< *options* > ;

In the same LIBNAME statement you can use any combination of specifications: librefs, physical pathnames, or a combination of librefs and pathnames. See Example 2 on page 918. Also see "Rules for Library Concatenation" on page 917.

Comparisons

Use the LIBNAME statement to reference a SAS data library or a DBMS. Use the FILENAME statement to reference an external file, such as a text or ASCII file you are reading data from or writing a report to.

Examples

Example 1: Specifying a LIBNAME Statement to Access ORACLE Data

In this example, the libref MYLIB uses the ORACLE engine to connect to an ORACLE database. The *SAS/ACCESS-engine-connection-options* are USER=, PASSWORD=, and PATH=. PATH= specifies an alias for the ORACLE driver, node, and database names, as required by SQL*NET version 2.0 or later.

```
libname mylib oracle user=scott password=tiger
        path="blunzer:v7" schema=hrdept;

proc print data=mylib.all_employees;
   where state='CA';
run;
```

Example 2: Specifying SAS/ACCESS Information with Macros You can also specify the database engine name and connection options with macros. Here a DATA step view is created from the DB2 table, DEPT:

```
%let dbmseng= db2;
%let con = ssid=db2a server=servr7;
libname mylib &dbmseng &con connection=sharedread;

data myview2/view=myview2;
   set mylib.dept(drop=deptno);
   where balance > 10000;
run;
```

Note that you can specify the DROP= data set option after the DB2 table MYLIB.DEPT, just as you can specify DROP= after any SAS data set. The new DATA step view MYVIEW2 references the same columns as MYLIB.DEPT except for the dropped DEPTNO column.

Example 3: Joining Two DBMS Tables In this example, the SQL procedure is used to join two tables in a database that is accessed through ODBC. By using the DQUOTE= option in the PROC SQL statement, you can specify and rename DBMS column names that otherwise would not be valid SAS names.

```
%let dbmseng = odbc;
%let con = user=josuha password=freude
           datasrc="Jo's Data";

libname dbms1 &dbmseng &con;

proc sql dquote=ansi;
   select first.work_id, first."@lastname" as lastname,
          second."birth date" as birthdate
      from  dbms1.employees1 as first,
            dbms1.employees2 as second
      where first.work_id=second.work_id;
```

The SQL procedure has many enhancements in Version 7. For more information, see the SAS/ACCESS documentation and "The SQL Procedure" in the *SAS Procedures Guide.*

See Also

Statement:

"LIBNAME" on page 913

SAS/ACCESS documentation for your DBMS

LINK

Jumps to a statement label

Valid: in a DATA step
Category: Control
Type: Executable

Syntax

LINK *label*;

Arguments

label
specifies a statement label that identifies the LINK destination. You must specify the *label* argument.

Details

The LINK statement tells SAS to jump immediately to the statement label that is indicated in the LINK statement and to continue executing statements from that point until a RETURN statement is executed. The RETURN statement sends program control to the statement immediately following the LINK statement.

The LINK statement and the destination must be in the same DATA step. The destination is identified by a statement label in the LINK statement.

The LINK statement can branch to a group of statements that contains another LINK statement. This arrangement is known as nesting. To avoid infinite looping, SAS has set a maximum on the number of nested LINK statements. Therefore, you can have up to ten LINK statements with no intervening RETURN statements. When more than one LINK statement has been executed, a RETURN statement tells SAS to return to the statement that follows the last LINK statement that was executed.

Comparisons

The difference between the LINK statement and the GO TO statement is in the action of a subsequent RETURN statement. A RETURN statement after a LINK statement returns execution to the statement that follows LINK. A RETURN statement after a GO TO statement returns execution to the beginning of the DATA step, unless a LINK statement precedes GO TO, in which case execution continues with the first statement after LINK. In addition, a LINK statement is usually used with an explicit RETURN statement, whereas a GO TO statement is often used without a RETURN statement.

When your program executes a group of statements at several points in the program, using the LINK statement simplifies coding and makes program logic easier to follow. If your program executes a group of statements at only one point in the program, using DO-group logic rather than LINK-RETURN logic is simpler.

Examples

In this example, when the value of variable TYPE is **aluv**, the LINK statement diverts program execution to the statements that are associated with the label CALCU. The program executes until it encounters the RETURN statement, which sends program execution back to the first statement that follows LINK. SAS executes the assignment statement, writes the observation, and then returns to the top of the DATA step to read the next record. When the value of TYPE is not **aluv**, SAS executes the assignment statement, writes the observation, and returns to the top of the DATA step.

```
data hydro;
   input type $ depth station $;
      /* link to label calcu: */
   if type ='aluv' then link calcu;
   date=today();
      /* return to top of step */
   return;
   calcu: if station='site_1'
      then elevatn=6650-depth;
   else if station='site_2'
      then elevatn=5500-depth;
         /* return to date=today(); */
   return;
   datalines;
aluv 523 site_1
uppa 234 site_2
aluv 666 site_2
...more data lines...
;
```

See Also

Statements:

"DO" on page 789
"GO TO" on page 845
"Labels, Statement" on page 908
"RETURN" on page 1004

LIST

Writes to the SAS log the input data records for the observation that is being processed

Valid: in a DATA step
Category: Action
Type: Executable

Syntax

LIST;

Without Arguments

The LIST statement causes the input data records for the observation being processed to be written to the SAS log.

Details

The LIST statement operates only on data that are read with an INPUT statement; it has no effect on data that are read with a SET, MERGE, MODIFY, or UPDATE statement.

In the SAS log, a ruler that indicates column positions appears before the first record listed.

Comparisons

Action	LIST statement	PUT statement
Writes when	at the end of each iteration of the DATA step	immediately
Writes what	the input data records exactly as they appear	the variables or literals specified
Writes where	only to the SAS log	to the SAS log, the SAS output destination, or to any external file

Action	LIST statement	PUT statement
Works with	INPUT statement only	any data-reading statement
Handles Hex Values	automatically prints a hexadecimal value if it encounters an unprintable character	represents characters in hexadecimal only when a hex format is given

Examples

This example uses the LIST statement to write to the SAS log any input records that contain missing data.

```
data employee;
   input ssn 1-9 #3 w2amt 1-6;
   if w2amt=. then list;
   datalines;
23456789
JAMES SMITH
356.79
345671234
Jeffrey Thomas
.
;
```

Because of the #3 line pointer control in the INPUT statement, SAS reads three input records to create a single observation. Therefore, the LIST statement writes the three current input records to the SAS log each time a value for W2AMT is missing:

Output 6.8

```
RULE:----+----1----+----2----+----3----+----4----+----5----+----
9  345671234
10  Jeffrey Thomas
11  .
```

The numbers 9, 10, and 11 are line numbers in the SAS log.

See Also

Statement:

"PUT" on page 962

%LIST

Displays lines that are entered in the current session

Valid: anywhere

Category: Program Control

Syntax

%LIST<*n* <:*m* | – *m*>>;

Without Arguments

In interactive line mode processing, if you use the %LIST statement without arguments, it displays all previously entered program lines.

Arguments

n
displays line *n*.

n– *m*
displays lines *n* through *m*.

Alias: *n*:*m*

Details

Where and When to Use The %LIST statement can be used anywhere in a SAS job except between a DATALINES or DATALINES4 statement and the matching semicolon (;) or semicolons (;;;;). This statement is useful mainly in interactive line mode sessions to display SAS program code on the monitor. It is also useful to determine lines to include when you use the %INCLUDE statement.

Interactions

CAUTION:
In all modes of execution, the SPOOL system option controls whether SAS statements are saved.

When the SPOOL system option is in effect in interactive line mode, all SAS statements and data lines are saved automatically when they are submitted. You can display them by using the %LIST statement. When NOSPOOL is in effect, %LIST cannot display previous lines. △

Examples

This %LIST statement displays lines 10 through 20:

```
%list 10-20;
```

See Also

Statement:

"%INCLUDE" on page 851

System Option:

"SPOOL" on page 1161

LOSTCARD

Resynchronizes the input data when SAS encounters a missing or invalid record in data that have multiple records per observation

Valid: in a DATA step

Category: Action

Type: Executable

Syntax

LOSTCARD;

Without Arguments

The LOSTCARD statement prevents SAS from reading a record from the next group when the current group has a missing record.

Details

When to Use LOSTCARD When SAS reads multiple records to create a single observation, it does not discover that a record is missing until it reaches the end of the data. If there is a missing record in your data, the values for subsequent observations in the SAS data set may be incorrect. Using LOSTCARD prevents SAS from reading a record from the next group when the current group has fewer records than SAS expected.

LOSTCARD is most useful when the input data have a fixed number of records per observation and when each record for an observation contains an identification variable that has the same value. LOSTCARD usually appears in conditional processing, for example, in the THEN clause of an IF-THEN statement, or in a statement in a SELECT group.

When LOSTCARD Executes When LOSTCARD executes, SAS takes these steps:

1. Writes three items to the SAS log: a lost card message, a ruler, and all the records that it read in its attempt to build the current observation.
2. Discards the first record in the group of records being read, does not write an observation, and returns processing to the beginning of the DATA step.
3. Does not increment the automatic variable _N_ by 1. (Normally, SAS increments _N_ by 1 at the beginning of each DATA step iteration.)
4. Attempts to build an observation by beginning with the second record in the group, and reads the number of records that the INPUT statement specifies.

5 Repeats steps 1 through 4 when the IF condition for a lost card is still true. To make the log more readable, SAS prints the message and ruler only once for a given group of records. In addition, SAS prints each record only once, even if a record is used in successive attempts to build an observation.

6 Builds an observation and writes it to the SAS data set when the IF condition for a lost card is no longer true.

Examples

This example uses the LOSTCARD statement in a conditional construct to identify missing data records and to resynchronize the input data:

```
data inspect;
   input id 1-3 age 8-9 #2 id2 1-3 loc
         #3 id3 1-3 wt;
   if id ne id2 or id ne id3 then
    do;
      put 'DATA RECORD ERROR: ' id= id2= id3=;
      lostcard;
    end;
   datalines;
301    32
301    61432
301    127
302    61
302    83171
400    46
409    23145
400    197
411    53
411    99551
411    139
;
```

The DATA step reads three input records before writing an observation. If the identification number in record 1 (variable ID) does not match the identification number in the second record (ID2) or third record (ID3), a record is incorrectly entered or omitted. The IF-THEN DO statement specifies that if an identification number is invalid, SAS prints the message that is specified in the PUT statement message and executes the LOSTCARD statement.

In this example, the third record for the second observation (ID3=400) is missing. The second record for the third observation is incorrectly entered (ID=400 while ID2=409). Therefore, the data set contains two observations with ID values 301 and 411. There are no observations for ID=302 or ID=400. The PUT and LOSTCARD statements write these statements to the SAS log when the DATA step executes:

Output 6.9

```
DATA RECORD ERROR: id=302 id2=302 id3=400
NOTE: LOST CARD.
RULE:----+----1----+----2----+----3----+----4----+----5----+----
4   302    61
5   302    83171
6   400    46
DATA RECORD ERROR: id=302 id2=400 id3=409
NOTE: LOST CARD.
7   409    23145
DATA RECORD ERROR: id=400 id2=409 id3=400
NOTE: LOST CARD.
8   400    197
DATA RECORD ERROR: id=409 id2=400 id3=411
NOTE: LOST CARD.
9   411    53
DATA RECORD ERROR: id=400 id2=411 id3=411
NOTE: LOST CARD.
20   411    99551
```

The numbers 14, 15, 16, 17, 18, 19, and 20 are line numbers in the SAS log.

See Also

Statement:

"IF-THEN/ELSE" on page 849

MERGE

Joins observations from two or more SAS data sets into single observations

Valid: in a DATA step
Category: File-handling
Type: Executable

Syntax

MERGE *SAS-data-set-1* <(*data-set-options*)>
SAS-data-set-2 <(*data-set-options*) >
<. . . *SAS-data-set-n*<(*data-set-options*)>>
<END=*variable*>;

Arguments

SAS-data-set(s)
names at least two existing SAS data sets from which observations are read.

Tip: Optionally, you can specify additional SAS data sets.

(*data-set-options*)
specifies one or more SAS data set options in parentheses after a SAS data set name.

Explanation: The data set options specify actions that SAS is to take when it reads observations into the DATA step for processing. For a list of data set options, see Chapter 2, "Data Set Options," on page 5.

END=*variable*
names and creates a temporary variable that contains an end-of-file indicator.

Explanation: The variable, which is initialized to 0, is set to 1 when the MERGE statement processes the last observation. If the input data sets have different numbers of observations, the END= variable is set to 1 when MERGE processes the last observation from all data sets.

Tip: The END= variable is not added to any SAS data set that is being created.

Details

Overview The MERGE statement is flexible and has a variety of uses in SAS programming. This section describes basic uses of MERGE. Other applications include using more than one BY variable, merging more than two data sets, and merging a few observations with all observations in another data set.

One-to-One Merging One-to-one merging combines observations from two or more SAS data sets into a single observation in a new data set. To perform a one-to-one merge, use the MERGE statement without a BY statement. SAS combines the first observation from all data sets that are named in the MERGE statement into the first observation in the new data set, the second observation from all data sets into the second observation in the new data set, and so on. In a one-to-one merge, the number of observations in the new data set is equal to the number of observations in the largest data set named in the MERGE statement. See Example 1 for an example of a one-to-one merge. For more information, see "Reading, Modifying, and Combining SAS Data Sets" in *SAS Language Reference: Concepts*.

CAUTION:
Use care when you combine data sets with a one-to-one merge. One-to-one merges may sometimes produce undesirable results. Test your program on representative samples of the data sets before you use this method. △

Match-Merging Match-merging combines observations from two or more SAS data sets into a single observation in a new data set according to the values of a common variable. The number of observations in the new data set is the sum of the largest number of observations in each BY group in all data sets. To perform a match-merge, use a BY statement immediately after the MERGE statement. The variables in the BY statement must be common to all data sets. Only one BY statement can accompany each MERGE statement in a DATA step. The data sets that are listed in the MERGE statement must be sorted in order of the values of the variables that are listed in the BY statement, or they must have an appropriate index. See Example 2 for an example of a match-merge. For more information, see "Reading, Modifying, and Combining SAS Data Sets" in *SAS Language Reference: Concepts*.

Comparisons

- MERGE combines observations from two or more SAS data sets. UPDATE combines observations from exactly two SAS data sets. UPDATE changes or updates the values of selected observations in a master data set as well. UPDATE also may add observations.
- Like UPDATE, MODIFY combines observations from two SAS data sets by changing or updating values of selected observations in a master data set.

- The results that are obtained by reading observations using two or more SET statements are similar to those that are obtained by using the MERGE statement with no BY statement. However, with the SET statements, SAS stops processing before all observations are read from all data sets if the number of observations are not equal. In contrast, SAS continues processing all observations in all data sets named in the MERGE statement.

Examples

Example 1: One-to-One Merging This example shows how to combine observations from two data sets into a single observation in a new data set:

```
data benefits.qtr1;
   merge benefits.jan benefits.feb;
run;
```

Example 2: Match-Merging This example shows how to combine observations from two data sets into a single observation in a new data set according to the values of a variable that is specified in the BY statement:

```
data inventry;
   merge stock orders;
   by partnum;
run;
```

See Also

Statements:

"BY" on page 765
"MODIFY" on page 933
"SET" on page 1010
"UPDATE" on page 1023

Combining SAS Data Sets in *SAS Language Reference: Concepts*

MISSING

Assigns characters in your input data to represent special missing values for numeric data

Valid: Anywhere
Category: Information

Syntax

MISSING *character(s)*;

Arguments

character
is the value in your input data that represents a special missing value.

Range: Special missing values can be any of the 26 letters of the alphabet (uppercase or lowercase) or the underscore (_).

Tip: You can specify more than one character.

Details

The MISSING statement usually appears within a DATA step, but it is global in scope.

Comparisons

The MISSING= system option allows you to specify a character to be printed when numeric variables contain ordinary missing values (.). If your data contain characters that represent special missing values, such as **a** or **z**, do not use the MISSING= option to define them; simply define these values in a MISSING statement.

Examples

With survey data, you may want to identify certain kinds of missing data. For example, in the data, an **A** can mean that the respondent is not at home at the time of the survey; an **R** can mean that the respondent refused to answer. Use the MISSING statement to identify to SAS that the values **A** and **R** in the input data lines are to be considered special missing values rather than invalid numeric data values:

```
data survey;
   missing a r;
   input id answer;
   datalines;
001 2
002 R
003 1
004 A
005 2
;
```

The resulting data set SURVEY contains exactly the values that are coded in the input data.

See Also

Statement:

"UPDATE" on page 1023

System Option:

"MISSING=" on page 1117

MODIFY

Replaces, deletes, and appends observations in an existing SAS data set in place; does not create an additional copy

Valid: in a DATA step

Category: File-handling

Type: Executable

Restriction: Cannot modify the descriptor portion of a SAS data set, such as adding a variable

Syntax

❶**MODIFY** *master-data-set* <(*data-set-option(s)*)> *transaction-data-set* <(*data-set-option(s)*)>
<NOBS=*variable*> <END=*variable*> <UPDATEMODE=MISSINGCHECK | NOMISSINGCHECK>;
BY *by-variable*;

❷**MODIFY** *master-data-set* <(*data-set-option(s)*)> KEY=*index* </ UNIQUE>
<NOBS=*variable*> <END=*variable*> ;

❸**MODIFY** *master-data-set* <(*data-set-option(s)*)> <NOBS=*variable*> POINT=*variable*;

❹**MODIFY** *master-data-set* <(*data-set-option(s)*)> <NOBS=*variable*> <END=*variable*>;

CAUTION:
Damage to the SAS data set can occur if the system terminates abnormally during a DATA step that contains the MODIFY statement. Observations in native SAS data files may have incorrect data values, or the data file may become unreadable. DBMS tables that are referenced by views are not affected. △

Note: If you modify a password-protected data set, specify the password with the appropriate data set option (ALTER= or PW=) within the MODIFY statement, and not in the DATA statement. △

Arguments

master-data-set
specifies the SAS data set that you want to modify.

Restriction: This data set must also appear in the DATA statement.

Restriction: For sequential and matching access, the master data set can be a SAS data file, a SAS/ACCESS view, an SQL view, or a DBMS engine for the LIBNAME statement. It cannot be a DATA step view or a passthrough view.

For random access using POINT=, the master data set must be a SAS data file or an SQL view that references a SAS data file.

For direct access using KEY=, the master data set can be a SAS data file or the DBMS engine for the LIBNAME statement. If it is a SAS file, it must be indexed and the index name must be specified on the KEY= option.

For a DBMS, the KEY= is set to the keyword DBKEY and the column names to use as an index must be specified on the DBKEY= data set option. These column names are used in constructing a WHERE expression that is passed to the DBMS.

transaction-data-set
specifies the SAS data set that provides the values for matching access. These values are the values that you want to use to update the master data set.

Restriction: Specify this data set *only* when the DATA step contains a BY statement.

END=*variable*
creates and names a temporary variable that contains an end-of-file indicator.

Explanation: The variable, which is initialized to zero, is set to 1 when the MODIFY statement reads the last observation of the data set being modified (for sequential access ❹) or the last observation of the transaction data set (for matching access ❶). It is also set to 1 when MODIFY cannot find a match for a KEY= value (random access ❷ ❸).

This variable is not added to any data set.

Restriction: Do not use this argument in the same MODIFY statement with the POINT= argument. POINT= indicates that MODIFY uses random access. The value of the END= variable is never set to 1 for random access.

KEY=*index*

names a simple or composite index of the SAS data file that is being modified. The KEY= argument retrieves observations from that SAS data file based on index values that are supplied by like-named variables in another source of information.

Default: If the KEY= value is not found, the automatic variable _ERROR_ is set to 1, and the automatic variable _IORC_ receives the value corresponding to the SYSRC autocall macro's mnemonic _DSENOM. See "Automatic Variable _IORC_ and the SYSRC Autocall Macro" on page 938 .

Restriction: KEY= processing is different for SAS/ACCESS engines. See the SAS/ACCESS documentation for more information.

Tip: Examples of sources for index values include a separate SAS data set named in a SET statement and an external file that is read by an INPUT statement.

Tip: If duplicates exist in the master file, only the first occurrence is updated unless you use a DO-LOOP to execute a SET statement for the data set listed on the KEY=option for all duplicates in the master data set.

If duplicates exist in the transaction data set, and they are consequtive, use the UNIQUE option to force the search for a match in the master data set to begin at the top of the index. Write an accumulation statement to add each duplicate transaction to the observation in master. Without the UNIQUE option, only the first duplicate transaction observation updates the master.

If the duplicates in the transaction data set are not consecutive, the search begins at the beginning of the index each time, so that each duplicate is applied to the master. Write an accumulation statement to add each duplicate to the master.

See Also: UNIQUE on page 936

Featured in: Example 4 on page 944, Example 5 on page 945, and Example 6 on page 946

NOBS=*variable*

creates and names a temporary variable whose value is usually the total number of observations in the input data set. For certain SAS views, SAS cannot determine the number of observations. In these cases, SAS sets the value of the NOBS= variable to the largest positive integer value available in the operating environment.

Explanation: At compilation time, SAS reads the descriptor portion of the data set and assigns the value of the NOBS= variable automatically. Thus, you can refer to the NOBS= variable before the MODIFY statement. The variable is available in the DATA step but is not added to the new data set.

Tip: The NOBS= and POINT= options are independent of each other.

Featured in: Example 3 on page 943

POINT=*variable*

reads SAS data sets using random (direct) access by observation number. *variable* names a variable whose value is the number of the observation to read. The POINT= variable is available anywhere in the DATA step, but it is not added to any SAS data set.

Requirement: When using the POINT= argument, include one or both of the following:

- a STOP statement
- programming logic that checks for an invalid value of the POINT= variable.

Because POINT= reads only the specified observations, SAS cannot detect an end-of-file condition as it would if the file were being read sequentially. Because detecting an end-of-file condition terminates a DATA step automatically, failure to substitute another means of terminating the DATA step when you use POINT= can cause the DATA step to go into a continuous loop.

Restriction: You cannot use the POINT= option with any of the following:

- BY statement
- WHERE statement
- WHERE= data set option
- transport format data sets
- sequential data sets (on tape or disk)
- a table from another vendor's relational database management system.

Restriction: You can use POINT= with compressed data sets only if the data set was created with the POINTOBS= data set option set to YES, the default value.

Restriction: You can use the random access method on compressed files only with SAS version 7 and beyond.

Tip: If the POINT= value does not match an observation number, SAS sets the automatic variable _ERROR_ to 1.

Featured in: Example 3 on page 943

UNIQUE

causes a KEY= search always to begin at the top of the index for the data file being modified.

Restriction: UNIQUE can appear only with the KEY= option.

Tip: Use UNIQUE when there are consecutive duplicate KEY= values in the transaction data set, so that the search for a match in the master data set begins at the top of the index file for each duplicate transaction. You must include an accumulation statement or the duplicate values overwrite each other causing only the last transaction value to be the result in the master observation.

Featured in: Example 5 on page 945

UPDATEMODE=MISSINGCHECK|
UPDATEMODE=NOMISSINGCHECK

specifies whether missing variable values in a transaction data set are to be allowed to replace existing variable values in a master data set.

MISSINGCHECK
prevents missing variable values in a transaction data set from replacing values in a master data set. Special missing values, however, are the exception and replace values in the master data set even when MISSINGCHECK is in effect.

NOMISSINGCHECK
allows missing variable values in a transaction data set to replace values in a master data set by preventing the check from being performed.

Default: MISSINGCHECK

Details

❶Matching Access The matching access method uses the BY statement to match observations from the transaction data set with observations in the master data set.

The BY statement specifies a variable that is in the transaction data set and the master data set.

When the MODIFY statement reads an observation from the transaction data set, it uses dynamic WHERE processing to locate the matching observation in the master data set. The observation in the master data set can be either

- replaced in the master data set with the value from the transaction data set
- deleted from the master data set
- appended to the master data set.

Example 2 on page 941 shows the matching access method.

❶Duplicate BY Values Duplicates in the master and transaction data sets affect processing.

- If duplicates exist in the master data set, only the first occurrence is updated because the generated WHERE statement always finds the first occurrence in the master.
- If duplicates exist in the transaction data set, the duplicates are applied one on top of another unless you write an accumulation statement to add all of them to the master observation. Without the accumulation statement, the values in the duplicates overwrite each other so that only the value in the last transaction is the result in the master observation.

❷Direct Access by Indexed Values This method requires that you use the KEY= option in the MODIFY statement to name an indexed variable from the data set that is being modified. Use another data source (typically a SAS data set named in a SET statement or an external file read by an INPUT statement) to provide a like-named variable whose values are supplied to the index. MODIFY uses the index to locate observations in the data set that is being modified.

Example 4 on page 944 shows the direct-access-by-indexed-values method.

❷Duplicate Index Values

- If there are duplicate values of the indexed variable in the master data set, only the first occurrence is retrieved, modified, or replaced. Use a DO LOOP to execute a SET statement with the KEY= option multiple times to update all duplicates with the transaction value.
- If there are duplicate, *nonconsecutive* values in the like-named variable in the data source, MODIFY applies each transaction cumulatively to the first observation in the master data set whose index value matches the values from the data source. Therefore, only the value in the last duplicate transaction is the result in the master observation unless you write an accumulation statement to accumulate each duplicate transaction value in the master observation.
- If there are duplicate, *consecutive* values in the variable in the data source, the values from the first observation in the data source are applied to the master data set, but the DATA step terminates with an error when it tries to locate an observation in the master data set for the second duplicate from the data source. To avoid this error, use the UNIQUE option in the MODIFY statement. The UNIQUE option causes SAS to return to the top of the master data set before retrieving a match for the index value. You must write an accumulation statement to accumulate the values from all the duplicates. If you do not, only the last one applied is the result in the master observation.

  Example 5 on page 945 shows how to handle duplicate index values.

- If there are duplicate index values in both data sets, you can use SQL to apply the duplicates in the transaction data set to the duplicates in the master data set in a one-to-one correspondence.

❸Direct (Random) Access by Observation Number You can use the POINT= option in the MODIFY statement to name a variable from another data source (not the master data set), whose value is the number of an observation that you want to modify in the master data set. MODIFY uses the values of the POINT= variable to retrieve observations in the data set that you are modifying. (You can use POINT= on a compressed data set only if the data set was created with the POINTOBS= data set option.)

It is good programming practice to validate the value of the POINT= variable and to check the status of the automatic variable _ERROR_.

Example 3 on page 943 shows the direct (random) access by observation number method.

CAUTION:

POINT= can result in infinite looping. Be careful when you use POINT=, as failure to terminate the DATA step can cause the DATA step to go into a continuous loop. Use a STOP statement, programming logic that checks for an invalid value of the POINT= variable, or both. △

❹Sequential Access The sequential access method is the simplest form of the MODIFY statement, but it provides less control than the direct access methods. With the sequential access method, you may use the NOBS= and END= options to modify a data set; you do not use the POINT= or KEY= options.

Automatic Variable _IORC_ and the SYSRC Autocall Macro The automatic variable _IORC_ contains the return code for each I/O operation that the MODIFY statement attempts to perform. The best way to test for values of _IORC_ is with the mnemonic codes that are provided by the SYSRC autocall macro. Each mnemonic code describes one condition. The mnemonics provide an easy method for testing problems in a DATA step program. These codes are useful:

_DSENMR
: specifies that the transaction data set observation does not exist on the master data set (used only with MODIFY and BY statements). If consecutive observations with different BY values do not find a match in the master data set, both of them return _DSENMR.

_DSEMTR
: specifies that multiple transaction data set observations with a given BY value do not exist on the master data set (used only with MODIFY and BY statements). If consecutive observations with the same BY values do not find a match in the master data set, the first observation returns _DSENMR and the subsequent observations return _DSEMTR.

_DSENOM
: specifies that the data set being modified does not contain the observation that is requested by the KEY= option or the POINT= option.

_SENOCHN
: specifies that SAS is attempting to execute an OUTPUT or REPLACE statement on an observation that contains a key value which duplicates one already existing on an indexed data set that requires unique key values.

_SOK
: specifies that the observation was located.

Note: Beginning in Version 7, the IORCMSG function returns a formatted error message associated with the current value of _IORC_. △

Example 6 on page 946 shows how to use the automatic variable _IORC_ and the SYSRC autocall macro.

Writing Observations When MODIFY Is Used in a DATA Step The way SAS writes observations to a SAS data set when the DATA step contains a MODIFY statement depends on whether certain other statements are present. The possibilities are

no explicit statement
: writes the current observation to its original place in the SAS data set. The action occurs as the last action in the step (as though a REPLACE statement were the last statement in the step).

OUTPUT statement
: if no data set is specified in the OUTPUT statement, writes the current observation to the end of all data sets that are specified in the DATA step. If a data set is specified, the statement writes the current observation to the end of the data set that is indicated. The action occurs at the point in the DATA step where the OUTPUT statement appears.

REPLACE <*data-set-name*> statement
: rewrites the current observation in the specified data set(s), or, if no argument is specified, rewrites the current observation in each data set specified on the DATA statement. The action occurs at the point of the REPLACE statement.

REMOVE <*data-set-name*> statement
: deletes the current observation in the specified data set(s), or, if no argument is specified, deletes the current observation in each data set specified on the DATA statement. The deletion may be a physical one or a logical one, depending on the characteristics of the engine that maintains the data set.

Keep in mind the following as you work with these statements:

- When no OUTPUT, REPLACE, or REMOVE statement is specified, the default action is REPLACE.
- The OUTPUT, REPLACE, and REMOVE statements are independent of each other. More than one statement can apply to the same observation, as long as the sequence is logical.
- Using OUTPUT, REPLACE, or REMOVE in a DATA step overrides the default replacement of observations. If you use any one of these statements in a DATA step, you must explicitly program each action that you want to take.
- If both an OUTPUT statement and a REPLACE or REMOVE statement execute on a given observation, perform the OUTPUT action last to keep the position of the observation pointer correct.

Example 7 on page 948 shows how to use the OUTPUT, REMOVE, and REPLACE statements to write observations.

Using MODIFY in a SAS/SHARE Environment In a SAS/SHARE environment, the MODIFY statement accesses an observation in update mode. That is, the observation is locked from the time MODIFY reads it until a REPLACE or REMOVE statement executes. At that point the observation is unlocked. It cannot be accessed until it is re-read with the MODIFY statement. The MODIFY statement opens the data set in update mode, but the control level is based on the statement used. For example, KEY=

and POINT= are member-level locking. Refer to *SAS/SHARE User's Guide* for more information.

Comparisons

- When you use a MERGE, SET, or UPDATE statement in a DATA step, SAS creates a new SAS data set. The data set descriptor of the new copy can be different from the old one (variables added or deleted, labels changed, and so on). When you use a MODIFY statement in a DATA step, however, SAS does not create a new copy of the data set. As a result, the data set descriptor cannot change.

  For information on DBMS replacement rules, see the SAS/ACCESS documentation.
- If you use a BY statement with a MODIFY statement, MODIFY works much like the UPDATE statement, except that
  - neither the master data set nor the transaction data set needs to be sorted or indexed. (The BY statement that is used with MODIFY triggers dynamic WHERE processing.)

    Note: Dynamic WHERE processing can be costly if the MODIFY statement modifies a SAS data set that is not in sorted order or has not been indexed. Having the master data set in sorted order or indexed and having the transaction data set in sorted order reduces processing overhead, especially for large files. △
  - both the master data set and the transaction data set can have observations with duplicate values of the BY variables. MODIFY treats the duplicates as described in "❶Duplicate BY Values" on page 937.
  - MODIFY cannot make any changes to the descriptor information of the data set as UPDATE can. Thus, it cannot add or delete variables, change variable labels, and so on.

Input Data Set for Examples

The examples modify the INVTY.STOCK data set. INVTY.STOCK contains these variables:

PARTNO
: is a character variable with a unique value identifying each tool number.

DESC
: is a character variable with the text description of each tool.

INSTOCK
: is a numeric variable with a value describing how many units of each tool the company has in stock.

RECDATE
: is a numeric variable containing the SAS date value that is the day for which INSTOCK values are current.

PRICE
: is a numeric variable with a value that describes the unit price for each tool.

In addition, INVTY.STOCK contains a simple index on PARTNO. This DATA step creates INVTY.STOCK:

```
libname invty 'SAS-data-library';
```

```
options yearcutoff= 1920;

data invty.stock(index=(partno));
   input PARTNO $ DESC $ INSTOCK @17
         RECDATE date7. @25 PRICE;
   format  recdate date7.;
   datalines;
K89R seal   34  27jul95 245.00
M4J7 sander 98  20jun95 45.88
LK43 filter 121 19may96 10.99
MN21 brace 43   10aug96 27.87
BC85 clamp 80   16aug96 9.55
NCF3 valve 198  20mar96 24.50
KJ66 cutter 6   18jun96 19.77
UYN7 rod  211   09sep96 11.55
JD03 switch 383 09jan97 13.99
BV1E timer 26   03jan97 34.50
;
```

Examples

Example 1: Modifying All Observations This example replaces the date on all records in the data set INVTY.STOCK with the current date. This code replaces the value of the variable RECDATE with the current date for all observations in INVTY.STOCK:

```
data invty.stock;
   modify invty.stock;
   recdate=today();
run;
```

A printing of INVTY.STOCK shows that RECDATE has been modified:

INVTY.STOCK 1

PARTNO	DESC	INSTOCK	RECDATE	PRICE
K89R	seal	34	14MAR97	245.00
M4J7	sander	98	14MAR97	45.88
LK43	filter	121	14MAR97	10.99
MN21	brace	43	14MAR97	27.87
BC85	clamp	80	14MAR97	9.55
NCF3	valve	198	14MAR97	24.50
KJ66	cutter	6	14MAR97	19.77
UYN7	rod	211	14MAR97	11.55
JD03	switch	383	14MAR97	13.99
BV1E	timer	26	14MAR97	34.50

The MODIFY statement opens INVTY.STOCK for update processing. SAS reads one observation of INVTY.STOCK for each iteration of the DATA step and performs any operations that the code specifies. In this case, the code replaces the value of RECDATE with the result of the TODAY function for every iteration of the DATA step. An implicit REPLACE statement at the end of the step writes each observation to its previous location in INVTY.STOCK.

Example 2: Modifying Observations Using a Transaction Data Set This example adds the quantity of newly received stock to its data set INVTY.STOCK as well as updating the date on which stock was received. The transaction data set ADDINV in the WORK library contains the new data.

The ADDINV data set is the data set that contains the updated information. ADDINV contains these variables:

PARTNO
is a character variable that corresponds to the indexed variable PARTNO in INVTY.STOCK.

NWSTOCK
is a numeric variable that represents quantities of newly received stock for each tool.

ADDINV is the second data set in the MODIFY statement. SAS uses it as the transaction data set and reads each observation from ADDINV sequentially. Because the BY statement specifies the common variable PARTNO, MODIFY finds the first occurrence of the value of PARTNO in INVTY.STOCK that matches the value of PARTNO in ADDINV. For each observation with a matching value, the DATA step changes the value of RECDATE to today's date and replaces the value of INSTOCK with the sum of INSTOCK and NWSTOCK (from ADDINV). MODIFY does not add NWSTOCK to the INVTY.STOCK data set because that would modify the data set descriptor. Thus, it is not necessary to put NWSTOCK in a DROP statement.

This example specifies ADDINV as the transaction data set that contains information to modify INVTY.STOCK. A BY statement specifies the shared variable whose values locate the observations in INVTY.STOCK.

This DATA step creates ADDINV:

```
data addinv;
   input PARTNO $ NWSTOCK;
   datalines;
K89R 55
M4J7 21
LK43 43
MN21 73
BC85 57
NCF3 90
KJ66 2
UYN7 108
JD03 55
BV1E 27
;
```

This DATA step uses values from ADDINV to update INVTY.STOCK.

```
libname invty 'SAS-data-library';


data invty.stock;
   modify invty.stock addinv;
   by partno;
   RECDATE=today();
   INSTOCK=instock+nwstock;
   if _iorc_=0 then replace;
run;
```

A printing of INVTY.STOCK shows that INSTOCK and RECDATE have been modified:

INVTY.STOCK 1

PARTNO	DESC	INSTOCK	RECDATE	PRICE
K89R	seal	89	14MAR97	245.00
M4J7	sander	119	14MAR97	45.88
LK43	filter	164	14MAR97	10.99
MN21	brace	116	14MAR97	27.87
BC85	clamp	137	14MAR97	9.55
NCF3	valve	288	14MAR97	24.50
KJ66	cutter	8	14MAR97	19.77
UYN7	rod	319	14MAR97	11.55
JD03	switch	438	14MAR97	13.99
BV1E	timer	53	14MAR97	34.50

Example 3: Modifying Observations Located by Observation Number This example reads the data set NEWP, determines which observation number in INVTY.STOCK to update based on the value of TOOL_OBS, and performs the update. This example explicitly specifies the update activity by using an assignment statement to replace the value of PRICE with the value of NEWP.

The data set NEWP contains two variables:

TOOL_OBS
contains the observation number of each tool in the tool company's master data set, INVTY.STOCK.

NEWP
contains the new price for each tool.

This DATA step creates NEWP:

```
data newp;
   input TOOL_OBS NEWP;
   datalines;
 251.00
2 49.33
3 12.32
4 30.00
5 15.00
6 25.75
7 22.00
8 14.00
9 14.32
0 35.00
;
```

This DATA step updates INVTY.STOCK:

```
libname invty 'SAS-data-library';


data invty.stock;
   set newp;
   modify invty.stock point=tool_obs
          nobs=max_obs;
   if _error_=1 then
      do;
```

```
        put 'ERROR occurred for TOOL_OBS=' tool_obs /
        'during DATA step iteration' _n_ /
        'TOOL_OBS value may be out of range.';
        _error_=0;
        stop;
      end;
   PRICE=newp;
   RECDATE=today();
run;
```

A printing of INVTY.STOCK shows that RECDATE and PRICE have been modified:

INVTY.STOCK 1

PARTNO	DESC	INSTOCK	RECDATE	PRICE
K89R	seal	34	14MAR97	251.00
M4J7	sander	98	14MAR97	49.33
LK43	filter	121	14MAR97	12.32
MN21	brace	43	14MAR97	30.00
BC85	clamp	80	14MAR97	15.00
NCF3	valve	198	14MAR97	25.75
KJ66	cutter	6	14MAR97	22.00
UYN7	rod	211	14MAR97	14.00
JD03	switch	383	14MAR97	14.32
BV1E	timer	26	14MAR97	35.00

Example 4: Modifying Observations Located by an Index This example uses the KEY= option to identify observations to retrieve by matching the values of PARTNO from ADDINV with the indexed values of PARTNO in INVTY.STOCK. ADDINV is created in Example 2 on page 941.

KEY= supplies index values that allow MODIFY to access directly the observations to update. No dynamic WHERE processing occurs. In this example, you specify that the value of INSTOCK in the master data set INVTY.STOCK increases by the value of the variable NWSTOCK from the transaction data set ADDINV.

```
libname invty 'SAS-data-library';


data invty.stock;
   set addinv;
   modify invty.stock key=partno;
   INSTOCK=instock+nwstock;
   RECDATE=today();
   if _iorc_=0 then replace;
run;
```

A printing of INVTY.STOCK shows that INSTOCK and RECDATE have been modified.

INVTY.STOCK 1

PARTNO	DESC	INSTOCK	RECDATE	PRICE
K89R	seal	89	14MAR97	245.00
M4J7	sander	119	14MAR97	45.88
LK43	filter	164	14MAR97	10.99
MN21	brace	116	14MAR97	27.87
BC85	clamp	137	14MAR97	9.55
NCF3	valve	288	14MAR97	24.50
KJ66	cutter	8	14MAR97	19.77
UYN7	rod	319	14MAR97	11.55
JD03	switch	438	14MAR97	13.99
BV1E	timer	53	14MAR97	34.50

Example 5: Handling Duplicate Index Values This example shows how MODIFY handles duplicate values of the variable in the SET data set that is supplying values to the index on the master data set.

The NEWINV data set is the data set that contains the updated information. NEWINV contains these variables:

PARTNO
is a character variable that corresponds to the indexed variable PARTNO in INVTY.STOCK. The NEWINV data set contains duplicate values for PARTNO; **M4J7** appears twice.

NWSTOCK
is a numeric variable that represents quantities of newly received stock for each tool.

This DATA step creates NEWINV:

```
data newinv;
   input PARTNO $ NWSTOCK;
   datalines;
K89R 55
M4J7 21
M4J7 26
LK43 43
MN21 73
BC85 57
NCF3 90
KJ66 2
UYN7 108
JD03 55
BV1E 27
;
```

This DATA step terminates with an error when it tries to locate an observation in INVTY.STOCK to match with the second occurrence of **M4J7** in NEWINV:

```
libname invty 'SAS-data-library';

/* This DATA step terminates with an error! */
data invty.stock;
   set newinv;
   modify invty.stock key=partno;
   INSTOCK=instock+nwstock;
```

```
   RECDATE=today();
run;
```

This message appears in the SAS log:

```
ERROR: No matching observation was found in MASTER data set.
PARTNO=K89R NWSTOCK=55 DESC=  INSTOCK=. RECDATE=14MAR97 PRICE=.
_ERROR_=1 _IORC_=1230015 _N_=1
NOTE: Missing values were generated as a result of performing
      an operation on missing values.
      Each place is given by:
      (Number of times) at (Line):(Column).
      1 at 689:19
NOTE: The SAS System stopped processing this step because of
      errors.
NOTE: The data set INVTY.STOCK has been updated.  There were 0
      observations rewritten, 0 observations added and 0
      observations deleted.
```

Adding the UNIQUE option to the MODIFY statement avoids the error in the previous DATA step. The UNIQUE option causes the DATA step to return to the top of the index each time it looks for a match for the value from the SET data set. Thus, it finds the **M4J7** in the MASTER data set for each occurrence of **M4J7** in the SET data set. The updated result for **M4J7** in the output shows that both values of NWSTOCK from NEWINV for **M4J7** are added to the value of INSTOCK for **M4J7** in INVTY.STOCK. An accumulation statement sums the values; without it, only the value of the last instance of **M4J7** would be the result in INVTY.STOCK.

```
data invty.stock;
   set newinv;
   modify invty.stock key=partno / unique;
   INSTOCK=instock+nwstock;
   RECDATE=today();
   if _iorc_=0 then replace;
run;
```

A printing of INVTY.STOCK shows that INSTOCK and RECDATE have been modified:

```
            Results of Using the UNIQUE Option    1

      PARTNO    DESC      INSTOCK    RECDATE     PRICE

       K89R     seal          89     14MAR97    245.00
       M4J7     sander       145     14MAR97     45.88
       LK43     filter       164     14MAR97     10.99
       MN21     brace        116     14MAR97     27.87
       BC85     clamp        137     14MAR97      9.55
       NCF3     valve        288     14MAR97     24.50
       KJ66     cutter         8     14MAR97     19.77
       UYN7     rod          319     14MAR97     11.55
       JD03     switch       438     14MAR97     13.99
       BV1E     timer         53     14MAR97     34.50
```

Example 6: Controlling I/O This example uses the SYSRC autocall macro and the _IORC_ automatic variable to control I/O condition. This technique helps to prevent unexpected results that could go undetected. This example uses the direct access

method with an index to update INVTY.STOCK. The data in the NEWSHIP data set updates INVTY.STOCK.

This DATA step creates NEWSHIP:

```
options yearcutoff= 1920;

data newship;
   input PARTNO $ DESC $ NWSTOCK @17
         SHPDATE date7. @25 NWPRICE;
   datalines;
K89R seal 14    14nov96 245.00
M4J7 sander 24  23aug96 47.98
LK43 filter 11  29jan97 14.99
MN21 brace 9    09jan97 27.87
BC85 clamp 12   09dec96 10.00
ME34 cutter 8   14nov96 14.50
;
```

Each WHEN clause in the SELECT statement specifies actions for each input/output return code that is returned by the SYSRC autocall macro:

- _SOK indicates that the MODIFY statement executed successfully.
- _DSENOM indicates that no matching observation was found in INVTY.STOCK. The OUTPUT statement specifies that the observation be appended to INVTY.STOCK. See the last observation in the output.
- If any other code is returned by SYSRC, the DATA step terminates and the PUT statement writes the message to the log.

```
libname invty 'SAS-data-library';

data invty.stock;
   set newship;
   modify invty.stock key=partno;
   select (_iorc_);
      when (%sysrc(_sok)) do;
         INSTOCK=instock+nwstock;
         RECDATE=shpdate;
         PRICE=nwprice;
         replace;
      end;
      when (%sysrc(_dsenom)) do;
         INSTOCK=nwstock;
         RECDATE=shpdate;
         PRICE=nwprice;
         output;
         _error_=0;
      end;
      otherwise do;
         put
         'An unexpected I/O error has occurred.'/
         'Check your data and your program';
         _error_=0;
         stop;
       end;
   end;
```

```
run;
```

```
                    INVTY.STOCK Data Set           1

       PARTNO    DESC       INSTOCK    RECDATE     PRICE

        K89R     seal         48       14NOV96    245.00
        M4J7     sander      122       23AUG96     47.98
        LK43     filter      132       29JAN97     14.99
        MN21     brace        52       09JAN97     27.87
        BC85     clamp        92       09DEC96     10.00
        NCF3     valve       198       20MAR96     24.50
        KJ66     cutter        6       18JUN96     19.77
        UYN7     rod         211       09SEP96     11.55
        JD03     switch      383       09JAN97     13.99
        BV1E     timer        26       03JAN97     34.50
        ME34     cutter        8       14NOV96     14.50
```

Example 7: Replacing and Removing Observations and Writing Observations to Different SAS Data Sets This example shows that you can replace and remove (delete) observations and write observations to different data sets. Further, this example shows that if an OUTPUT, REPLACE, or REMOVE statement is present, you must specify explicitly what action to take because no default statement is generated.

The parts that were received in 1997 are output to INVTY.STOCK97 and are removed from INVTY.STOCK. Likewise, the parts that were received in 1995 are output to INVTY.STOCK95 and are removed from INVTY.STOCK. Only the parts that were received in 1996 remain in INVTY.STOCK, and the PRICE is updated only in INVTY.STOCK.

```
libname invty 'SAS-data-library';

data invty.stock invty.stock95 invty.stock97;
   modify invty.stock;
   if recdate>'01jan97'd then do;
       output invty.stock97;
       remove invty.stock;
   end;
   else if recdate<'01jan96'd then do;
       output invty.stock95;
       remove invty.stock;
   end;
   else do;
      price=price*1.1;
      replace invty.stock;
   end;
run;
```

```
              New Prices for Stock Rcvd in '96     1

       PARTNO    DESC       INSTOCK    RECDATE     PRICE

        LK43     filter      121       19MAY96    12.089
        MN21     brace        43       10AUG96    30.657
        BC85     clamp        80       16AUG96    10.505
        NCF3     valve       198       20MAR96    26.950
        KJ66     cutter        6       18JUN96    21.747
        UYN7     rod         211       09SEP96    12.705
```

See Also

Statements:

"OUTPUT" on page 958
"REMOVE" on page 993
"REPLACE" on page 997
"UPDATE" on page 1023

"Reading, Combining, and Modifying SAS Data Sets" in *SAS Language Reference: Concepts*

"SQL Procedure" in the *SAS Procedures Guide*

Null

Signals the end of data lines; acts as a placeholder

Valid: Anywhere
Category: Action
Type: Executable

Syntax

;

or

;;;;

Without Arguments

The Null statement signals the end of the data lines that occur in your program.

Details

The primary use of the Null statement is to signal the end of data lines that follow a DATALINES or CARDS statement. In this case, the Null statement functions as a step boundary. When your data lines contain semicolons, use the DATALINES4 or CARDS4 statement and a Null statement of four semicolons.

Although the Null statement performs no action, it is an executable statement. Therefore, a label can precede the Null statement, or you can use it in conditional processing.

Examples

□ The Null statement in this program marks the end of data lines and functions as a step boundary.

```
data test;
   input score1 score2 score3;
   datalines;
```

```
55 135 177
44 132 169
;
```

- The input data records in this example contain semicolons. Use the Null statement following the DATALINES4 statement to signal the end of the data lines.

```
data test2;
   input code1 $ code2 $ code3 $;
   datalines4;
55;39;1  135;32;4  177;27;3
78;29;1  149;22;4  179;37;3
;;;;
```

- The Null statement is useful while you are developing a program. For example, use it after a statement label to test your program before you code the statements that follow the label.

```
data _null_;
   set dsn;
   file print header=header;
   put 'report text';
     ...more statements...
   return;
   header:;
run;
```

See Also

Statements:

"DATALINES" on page 781
"DATALINES4" on page 782
"GO TO" on page 845
"LABEL" on page 906

ODS EXCLUDE

Specifies output objects to exclude from ODS destinations

Valid: anywhere

Category: Output Control

Syntax

ODS <*ODS-destination*> **EXCLUDE** *exclusion(s)* | ALL | NONE;

To do this …	Use this argument
Specify which ODS destination's exclusion list to write to. The ODS destination can be HTML, LISTING, or PRINTER.	*ODS-destination*
Identify which output objects to add to an exclusion list.	*exclusion*
Set the list to EXCLUDE ALL.	ALL
Set the list to EXCLUDE NONE (EXCLUDE NONE has the same effect as SELECT ALL.)	NONE

See

The Complete Guide to the SAS Output Delivery System for complete information.

ODS HTML

Opens, manages, or closes the HTML destination. If the destination is open, you can create HTML output (output that is written in Hypertext Markup Language).

Valid: anywhere

Category: Output Control

Syntax

ODS HTML *action*;

ODS HTML *HTML-file-specification(s)* <*option(s)*>;

To do this …	Use this *action*
Close the HTML destination and any files that are associated with it	CLOSE
Select output objects for the HTML destination	SELECT
Exclude output objects from the HTML destination	EXCLUDE
Write to the SAS log the current selection or exclusion list for the HTML destination	SHOW

To do this …	Use this option
Specify the base name for the HTML anchor tag that identifies each output object in the current body file	ANCHOR=
Specify a string to use as the first part of all links and references that ODS creates in the HTML files	BASE=
Control the destination of the footnotes that are defined by the graphics program that generates the HTML output	GFOOTNOTE | NOGFOOTNOTE
Specify the destination for all graphics output that is generated while the HTML destination is open	GPATH=
Control the destination of the titles that are defined by the graphics program that generates the HTML output	GTITLE | NOGTITLE
Specify HTML to place between the <HEAD> and </HEAD> tags in all the HTML files that the HTML destination writes to	HEADTEXT=
Specify HTML to use as the <META> tag inside the <HEAD> and </HEAD> tags of all the HTML files that the HTML destination writes to	METATEXT=
Create a new body file at the specified starting-point	NEWFILE=
Specify the location (an external file or a SAS catalog) for all HTML files	PATH=
Specify an alternative record separator	RECORD_SEPARATOR
Specify the style definition to use in writing the HTML files	STYLE=
Translate the HTML files to the requested represention	TRANTAB=

See

The Complete Guide to the SAS Output Delivery System for complete information.

ODS LISTING

Opens, manages, or closes the Listing destination

Valid: anywhere

Category: Output Control

Syntax

ODS LISTING <*action*>;

ODS LISTING <DATAPANEL=*number* | DATA | PAGE>;

To do this …	Use this *action*
Close the Listing destination	CLOSE
Select output objects for the Listing destination	SELECT
Exclude output objects from the Listing destination	EXCLUDE
Write to the SAS log the current selection or exclusion list for the Listing destination	SHOW

See

The Complete Guide to the SAS Output Delivery System for complete information.

ODS OUTPUT

Creates a SAS data set from an output object and manages the selection and exclusion lists for the Output destination

Valid: anywhere

Category: Output Control

Syntax

ODS OUTPUT *action*

ODS OUTPUT *data-set-definition(s)*;

To do this …	Use this *action*
Set the list for the Output destination to EXCLUDE ALL	CLEAR
Close the Output destination	CLOSE
Write to the SAS log the current selection or exclusion list for the Output destination	SHOW

To do this …	Use this argument
Provide instructions for turning an output object into a SAS data set	*data-set-definition*

See

The Complete Guide to the SAS Output Delivery System for complete information.

ODS PATH

Specifies which locations to search for definitions that were created by PROC EMPLATE, as well as the order in which to search for them

Valid: anywhere

Category: Output Control

Syntax

ODS PATH *location(s)*;

To do this ...	Use this argument
Specify one or more locations to search for definitions that were created by PROC TEMPLATE	*location*

See

The Complete Guide to the SAS Output Delivery System for complete information.

ODS PRINTER

Opens, manages, or closes the Printer destination. If the destination is open, you can create Printer output (output that is formatted for a high-resolution printer)

Valid: anywhere

Category: Output Control

Syntax

ODS PRINTER <*action*>;

ODS PRINTER <*option(s)*>;

To do this ...	Use this *action*
Close the Printer destination and the file that is associated with it	CLOSE
Select output objects for the Printer destination	SELECT
Exclude output objects from the Printer destination	EXCLUDE
Write to the SAS log the current selection or exclusion list for the Printer destination	SHOW

To do this …	Use this option
Specify whether or not to use all the color information that the style provides	COLOR
Specify the file to write to	FILE=
Specify a scaling factor to apply to all the font sizes that do not have an explicit unit of measure	FONTSCALE=
Specify that ODS use the generic postscript driver that SAS provides	POSTSCRIPT
Specify the name of the printer for which to format the Printer output	PRINTER=
Specify that ODS use the printer drivers that SAS provides	SAS
Specify the style definition to use in writing the Printer output	STYLE=
Ensure uniform column widths for all pages of Printer output	UNIFORM

See

The Complete Guide to the SAS Output Delivery System for complete information.

ODS SELECT

Specifies output objects for ODS destinations

Valid: anywhere
Category: Output Control

Syntax

ODS SELECT <*ODS-destination*>*selection(s)* | ALL | NONE;

To do this …	Use this argument
Specify which ODS destination's selection list to write to. The ODS destination can be HTML, LISTING, or PRINTER.	*ODS-destination*
Identify which output objects to add to a selection list.	*selection*
Set the list to SELECT ALL.	ALL
Set the list to SELECT NONE.	NONE

See

The Complete Guide to the SAS Output Delivery System for complete information.

ODS SHOW

Writes to the SAS log the specified selection or exclusion list

Valid: anywhere

Category: Output Control

Syntax

ODS <*ODS-destination*> **SHOW**;

To do this …	Use this argument
Specify which ODS destination's selection or exclusion list to write to the SAS log. The ODS destination can be HTML, LISTING, or PRINTER.	*ODS-destination*

See

The Complete Guide to the SAS Output Delivery System for complete information.

ODS TRACE

Writes to the SAS log a record of each output object that is created, or suppresses the writing of this record

Valid: anywhere

Category: Output Control

Syntax

ODS TRACE ON< / <*option(s)*>>;

ODS TRACE OFF;

To do this …	Use this argument
Turn on the writing of the trace record	ON
Turn off the writing of the trace record	OFF

To do this …	Use this option
Include the label path for the output object in the record	LABEL
Write the trace record to the Listing destination so that each part of the trace record immediately precedes the output object that it describes	LISTING

See

The Complete Guide to the SAS Output Delivery System for complete information.

ODS VERIFY

Prints or suppresses a warning that a style definition or a table definition that is used is not supplied by SAS Institute

Valid: anywhere
Category: Output Control

Syntax

ODS VERIFY <ON | OFF | ERROR>;

To do this …	Use this option
Print the warning and send output objects to open destinations	ON
Supress the warning	OFF
Print the warning and do not send output objects to open destinations	ERROR

See

The Complete Guide to the SAS Output Delivery System for complete information.

OPTIONS

Changes the value of one or more SAS system options

Valid: anywhere
Category: Program Control

Syntax

OPTIONS *option(s)*;

Arguments

option
specifies one or more SAS system options to be changed.

Details

The change that is made by the OPTIONS statement remains in effect for the rest of the job, session, SAS process, or until you issue another OPTIONS statement to change the options again. You can specify SAS system options through the OPTIONS statement, through the OPTIONS window, at SAS invocation, and at the initiation of a SAS process.

Note: If you want a particular group of options to be in effect for all your SAS jobs or sessions, store an OPTIONS statement in an autoexec file or list the system options in a configuration file or custom_option_set. △

An OPTIONS statement can appear at any place in a SAS program, except within data lines.

Operating Environment Information: The system options that are available depend on your operating environment. Also, the syntax that is used to specify a system option in the OPTIONS statement may be different from the syntax that is used at SAS invocation. For details, see the SAS documentation for your operating environment. △

Comparisons

The OPTIONS statement requires you to enter the complete statement including system option name and value, if necessary. The SAS OPTIONS window displays the options' names and settings in columns. To change a setting, type over the value that is displayed and press ENTER or RETURN.

Examples

This example suppresses the date that is normally printed in SAS output and sets a line size of 72:

```
options nodate linesize=72;
```

See Also

"Definition" on page 1048

OUTPUT

Writes the current observation to a SAS data set

Valid: in a DATA step
Category: Action
Type: Executable

Syntax

OUTPUT<*data-set-name(s)*>;

Without Arguments

Using OUTPUT without arguments causes the current observation to be written to all data sets that are named in the DATA statement.

Note: If a MODIFY statement is present, OUTPUT with no arguments writes the current observation to the end of the data set that is specified in the MODIFY statement. △

Arguments

data-set-name
specifies the name of a data set to which SAS writes the observation.

Restriction: All names specified in the OUTPUT statement must also appear in the DATA statement.

Tip: You can specify up to as many data sets in the OUTPUT statement as you specified in the DATA statement for that DATA step.

Details

When and Where the OUTPUT Statement Writes Observations The OUTPUT statement tells SAS to write the current observation to a SAS data set immediately, not at the end of the DATA step. If no data set name is specified in the OUTPUT statement, the observation is written to the data set or data sets that are listed in the DATA statement.

Implicit versus Explicit Output By default, every DATA step contains an implicit OUTPUT statement at the end of each iteration that tells SAS to write observations to the data set or data sets that are being created. Placing an explicit OUTPUT statement in a DATA step overrides the automatic output, and SAS adds an observation to a data set only when an explicit OUTPUT statement is executed. Once you use an OUTPUT statement to write an observation to any one data set, however, there is no implicit OUTPUT statement at the end of the DATA step. In this situation, a DATA step writes an observation to a data set only when an explicit OUTPUT executes. You can use the OUTPUT statement alone or as part of an IF-THEN or SELECT statement or in DO-loop processing.

When Using the MODIFY Statement When you use the MODIFY statement with the OUTPUT statement, the REMOVE and REPLACE statements override the implicit write action at the end of each DATA step iteration. See “Comparisons” on page 960 for more information. If both the OUTPUT statement and a REPLACE or REMOVE statement execute on a given observation, perform the output action last to keep the position of the observation pointer correct.

Comparisons

- OUTPUT writes observations to a SAS data set; PUT writes variable values or text strings to an external file or the SAS log.
- To control when an observation is written to a specified output data set, use the OUTPUT statement. To control which variables are written to a specified output data set, use the KEEP= or DROP= data set option in the DATA statement, or use the KEEP or DROP statement.
- When you use the OUTPUT statement with the MODIFY statement, the following items apply.
  - Using an OUTPUT, REPLACE, or REMOVE statement overrides the default write action at the end of a DATA step. (OUTPUT is the default action; REPLACE becomes the default action when a MODIFY statement is used.) If you use any of these statements in a DATA step, you must explicitly program output for the new observations that are added to the data set.
  - The OUTPUT, REPLACE, and REMOVE statements are independent of each other. More than one statement can apply to the same observation, as long as the sequence is logical.
  - If both an OUTPUT and a REPLACE or REMOVE statement execute on a given observation, perform the OUTPUT action last to keep the position of the observation pointer correct.

Examples

Example 1: Sample Uses of OUTPUT These examples show how you can use an OUTPUT statement:

```
/* writes the current observation */
   /* to a SAS data set              */
output;
```

```
/* writes the current observation */
   /* when a condition is true       */
if deptcode gt 2000 then output;
```

```
/* writes an observation to data  */
   /* set MARKUP when the PHONE      */
   /* value is missing               */
if phone=. then output markup;
```

Example 2: Creating Multiple Observations from Each Line of Input You can create two or more observations from each line of input data. This SAS program creates three observations in the data set RESPONSE for each observation in the data set SULFA:

```
data response(drop=time1-time3);
   set sulfa;
   time=time1;
   output;
   time=time2;
   output;
   time=time3;
   output;
run;
```

Example 3: Creating Multiple Data Sets from a Single Input File You can create more than one SAS data set from one input file. In this example, OUTPUT writes observations to two data sets, OZONE and OXIDES:

```
options yearcutoff= 1920;

data ozone oxides;
   infile file-specification;
   input city $ 1-15 date date9.
      chemical $ 26-27 ppm 29-30;
   if chemical='O3' then output ozone;
   else output oxides;
run;
```

Example 4: Creating One Observation from Several Lines of Input You can combine several input observations into one observation. In this example, OUTPUT creates one observation that totals the values of DEFECTS in the first ten observations of the input data set:

```
data discards;
   set gadgets;
   drop defects;
   reps+1;
   if reps=1 then total=0;
   total+defects;
   if reps=10 then do;
      output;
      stop;
   end;
run;
```

See Also

Statements:

"DATA" on page 774
"MODIFY" on page 933
"PUT" on page 962
"REMOVE" on page 993
"REPLACE" on page 997

PAGE

Skips to a new page in the SAS log

Valid: Anywhere
Category: Log Control

Syntax

PAGE;

Without Arguments

The PAGE statement skips to a new page in the SAS log.

Details

You can use the PAGE statement when you run SAS in a windowing environment, batch, or noninteractive mode. The PAGE statement itself does not appear in the log. When you run SAS in interactive line mode, PAGE may print blank lines to the display monitor (or altlog file).

See Also

Statement:

"LIST" on page 925

System Options:

"LINESIZE=" on page 1112
"PAGESIZE=" on page 1133

PUT

Writes lines to the SAS log, to the SAS procedure output file, or to an external file that is specified in the most recent FILE statement

Valid: in a DATA step
Category: File-handling
Type: Executable

Syntax

PUT <*specification(s)*><_ODS_><@|@@>;

Without Arguments

The PUT statement without arguments is called a *null PUT statement*.

The null PUT statement

- writes the current output line to the current file, even if the current output line is blank.
- releases an output line that is being held by a previous PUT statement with a trailing @.

See: "Using Line-hold Specifiers" on page 969

Featured in: Example 5 on page 975

Arguments

specification
specifies what is written, how it is written, and where it is written. This can include

variable
names the variable whose value is written.

Note: Beginning with Version 7, you can specify column-mapped Output Delivery System variables in the PUT statement. This functionality is described briefly here in _ODS_ on page 964, but documented more completely in "PUT, _ODS_" on page 989 . For more information, see *The Complete Guide to the SAS Output Delivery System*. △

(*variable-list*)
specifies a list of variables whose values are written.

Requirement: The (*format-list*) must follow the (*variable-list*).

See: "PUT, Formatted" on page 979

'character-string'
specifies a string of text, enclosed in quotation marks, to write.

Tip: To write a hexadecimal string in EBCDIC or ASCII, follow the ending quotation mark with an **x**.

See Also: "List Output" on page 967

Example: This statement writes HELLO when the hexadecimal string is converted to ASCII characters:

```
put '68656C6C6F'x;
```

*n**
specifies to repeat *n* times the subsequent character string.

Example: This statement writes a line of 132 underscores.

```
put 132*'_';
```

Featured in: Example 4 on page 975

pointer-control
moves the output pointer to a specified line or column in the output buffer.

See: "Column Pointer Controls" on page 964 and "Line Pointer Controls" on page 966

column-specifications
specifies which columns of the output line the values are written.

See: "Column Output" on page 967

Featured in: Example 2 on page 972

format.
specifies a format to use when the variable values are written.

See: "Formatted Output" on page 968

Featured in: Example 1 on page 971

(*format-list*)
specifies a list of formats to use when the values of the preceding list of variables are written.

Restriction: The (*format-list*) must follow the (*variable-list*).

See: "PUT, Formatted" on page 979

INFILE
writes the last input data record that is read either from the current input file or from the data lines that follow a DATELINES statement.

Tip: _INFILE_ is a pseudo-variable that references the current INPUT buffer. You can use this pseudo-variable in other SAS statements.

Tip: If the most recent INPUT statement uses line-pointer controls to read multiple input data records, PUT _INFILE_ writes only the record that the input pointer is positioned on.

Example: This PUT statement writes all the values of the first input data record:

```
input #3 score #1 name $ 6-23;
put _infile_;
```

Featured in: Example 6 on page 976

ALL
writes the values of all variables, which includes automatic variables, that are defined in the current DATA step by using named output.

See: "Named Output" on page 968

ODS
moves data values for all columns (as defined by the ODS option in the FILE statement) into a special buffer, from which it is eventually written to the data component. The ODS option in the FILE statement defines the structure of the data component that holds the results of the DATA step.

Restriction: Use _ODS_ only if you have previously specified the ODS option in the FILE statement.

Tip: You can use the _ODS_ specification in conjunction with variable specifications and column pointers, and it can appear anywhere in a PUT statement.

Interaction: _ODS_ writes data into a specific column only if a PUT statement has not already specified a variable for that column with a column pointer. That is, a variable specification for a column overrides the _ODS_ option.

See: "PUT, _ODS_" on page 989 and *The Complete Guide to the SAS Output Delivery System*

@|@@
holds an output line for the execution of the next PUT statement even across iterations of the DATA step. These line-hold specifiers are called *trailing* @ and *double trailing* @.

Restriction: The trailing @ or double trailing @ must be the last item in the PUT statement.

Tip: Use an @ or @@ to hold the pointer at its current location. The next PUT statement that executes writes to the same output line rather than to a new output line.

See:

Featured in: Example 5 on page 975

Column Pointer Controls

@n
moves the pointer to column *n*.

Range: a positive integer

Example: @15 moves the pointer to column 15 before the value of NAME is written:

```
put @15 name $10.;
```

Featured in: Example 2 on page 972 and Example 4 on page 975

@numeric-variable
moves the pointer to the column given by the value of *numeric-variable*.

Range: a positive integer

Tip: If *n* is not an integer, SAS truncates the decimal portion and uses only the integer value. If *n* is zero or negative, the pointer moves to column 1.

Example: The value of the variable A moves the pointer to column 15 before the value of NAME is written:

```
a=15;
put @a name $10.;
```

Featured in: Example 2 on page 972

@(expression)
moves the pointer to the column that is given by the value of *expression*.

Range: a positive integer

Tip: If the value of *expression* is not an integer, SAS truncates the decimal value and uses only the integer value. If it is zero, the pointer moves to column 1.

Example: The result of the expression moves the pointer to column 15 before the value of NAME is written:

```
b=5;
put @(b*3) name $10.;
```

+n
moves the pointer *n* columns.

Range: a positive integer or zero

Tip: If *n* is not an integer, SAS truncates the decimal portion and uses only the integer value.

Example: This statement moves the pointer to column 23, writes a value of LENGTH in columns 23 through 26, advances the pointer five columns, and writes the value of WIDTH in columns 32 through 35:

```
put @23 length 4. +5 width 4.;
```

+numeric-variable
moves the pointer the number of columns given by the value of *numeric-variable*.

Range: a positive or negative integer or zero

Tip: If *numeric-variable* is not an integer, SAS truncates the decimal value and uses only the integer value. If *numeric-variable* is negative, the pointer moves backward. If the current column position becomes less than 1, the pointer moves to column 1. If the value is zero, the pointer does not move. If the value is greater than the length of the output buffer, the current line is written out and the pointer moves to column 1 on the next line.

+(expression)
moves the pointer the number of columns given by *expression*.

Range: *expression* must result in an integer

Tip: If *expression* is not an integer, SAS truncates the decimal value and uses only the integer value. If *expression* is negative, the pointer moves backward. If the current column position becomes less than 1, the pointer moves to column 1. If the value is zero, the pointer does not move. If the value is greater than the length of the output buffer, the current line is written out and the pointer moves to column 1 on the next line.

Featured in: Example 2 on page 972

Line Pointer Controls

#n
moves the pointer to line *n*.

Range: a positive integer

Example: The #2 moves the pointer to the second line before the value of ID is written in columns 3 and 4:

```
put @12 name $10. #2 id 3-4;
```

#numeric-variable
moves the pointer to the line given by the value of *numeric-variable*.

Range: a positive integer.

Tip: If the value of *numeric-variable* is not an integer, SAS truncates the decimal value and uses only the integer value.

#(*expression*)
moves the pointer to the line that is given by the value of *expression*.

Range: *Expression* must result in a positive integer.

Tip: If the value of *expression* is not an integer, SAS truncates the decimal value and uses only the integer value.

/
advances the pointer to column 1 of the next line.

Example: The values for NAME and AGE are written on one line, and then the pointer moves to the second line to write the value of ID in columns 3 and 4:

```
put name age / id 3-4;
```

Featured in: Example 3 on page 973

OVERPRINT
causes the values that follow the keyword OVERPRINT to print on the most recently written output line.

Requirement: You must direct the output to a print file. Set the N= option in the FILE statement to 1 and direct the PUT statements to a print file.

Tip: OVERPRINT has no effect on lines that are written to a display.

Tip: Use OVERPRINT in combination with column pointer and line pointer controls to overprint text.

Example: This statement overprints underscores, starting in column 15, which underlines the title:

```
put @15 'Report Title' overprint
    @15 '____________';
```

Featured in: Example 4 on page 975

BLANKPAGE
advances the pointer to the first line of a new page, even when the pointer is positioned on the first line and the first column of a new page.

Tip: If the current output file contains carriage control characters, _BLANKPAGE_ produces output lines that contain the appropriate carriage control character.

Featured in: Example 3 on page 973

PAGE
advances the pointer to the first line of a new page. SAS automatically begins a new page when a line exceeds the current PAGESIZE= value.

Tip: If the current output file is a print file, _PAGE_ produces an output line that contains the appropriate carriage control character. _PAGE_ has no effect on a nonprint file.

Featured in: Example 3 on page 973

Details

When to Use PUT

Use the PUT statement to write lines to the SAS log, to the SAS procedure output file, or to an external file. If you do not execute a FILE statement before the PUT statement in the current iteration of a DATA step, SAS writes the lines to the SAS log. If you specify the PRINT option in the FILE statement, SAS writes the lines to the procedure output file.

The PUT statement can write lines that contain variable values, character strings, and hexadecimal character constants. With specifications in the PUT statement, you specify what to write, where to write it, and how to format it.

Output Styles

There are four ways to write variable values with the PUT statement:

- column
- list (simple and modified)
- formatted
- named.

A single PUT statement may contain any or all of the available output styles, depending on how you want to write lines.

Column Output With *column output*, the column numbers follow the variable in the PUT statement. These numbers indicate where in the line to write the value:

```
put name 6-15 age 17-19;
```

These lines are written to the SAS log*:

```
----+----1----+----2----+
     Peterson    21
     Morgan      17
```

The PUT statement writes values for NAME and AGE in the specified columns. See “PUT, Column” on page 977 for more information.

List Output With *list output*, list the variables and character strings in the PUT statement in the order that you want to write them. For example, this PUT statement

```
put name age;
```

writes the values for NAME and AGE to the SAS log:

```
----+----1----+----2----+
Peterson 21
Morgan 17
```

See “PUT, List” on page 983 for more information.

* The ruled line is for illustrative purposes only; the PUT statement does not generate it.

Formatted Output With *formatted output*, specify a SAS format or a user-written format after the variable name. The format gives instructions on how to write the variable value. Formats allow you to write in a nonstandard form, such as packed decimal, or numbers that contain special characters such as commas. For example, this PUT statement

```
put name $char10. age 2. +1 date mmddyy10.;
```

writes the values for NAME, AGE, and DATE to the SAS log:

```
----+----1----+----2----+
Peterson  21 07/18/1999
Morgan    17 11/12/1999
```

Using a pointer control of +1 inserts a blank space between the values of AGE and DATE. See “PUT, Formatted” on page 979 for more information.

Named Output With *named output*, list the variable name followed by an equal sign. For example, this PUT statement

```
put name= age=;
```

writes the values for NAME and AGE to the SAS log:

```
----+----1----+----2----+
name=Peterson age=21
name=Morgan age=17
```

See “PUT, Named” on page 987 for more information.

Using Multiple Output Styles in a Single PUT Statement

A PUT statement can combine any or all of the different output styles. For example,

```
put name 'on ' date mmddyy8. ' weighs '
    startwght +(-1) '.' idno= 40-45;
```

See Example 1 on page 971 for an explanation of the lines written to the SAS log.

When you combine different output styles, it is important to understand the location of the output pointer after each value is written. For more information on the pointer location, see .

Pointer Controls

As SAS writes values with the PUT statement, it keeps track of its position with a pointer. The PUT statement provides three ways to control the movement of the pointer:

column pointer controls
: reset the pointer's column position when the PUT statement starts to write the value to the output line.

line pointer controls
: reset the pointer's line position when the PUT statement writes the value to the output line.

line-hold specifiers
: hold a line in the output buffer so that another PUT statement can write to it. By default, the PUT statement releases the previous line and writes to a new line.

With column and line pointer controls, you can specify an absolute line number or column number to move the pointer or you can specify a column or line location that is

relative to the current pointer position. Table 6.7 on page 969 lists all pointer controls that are available with the PUT statement.

Table 6.7 Pointer Controls Available in the PUT Statement

Pointer Controls	Relative	Absolute
column pointer controls	+*n*	@*n*
	+*numeric-variable*	@*numeric-variable*
	+(*expression*)	@(*expression*)
line pointer controls	/ , _PAGE_ ,	#*n*
	BLANKPAGE	#*numeric-variable*
		#(*expression*)
	OVERPRINT	none
line-hold specifiers	@	(not applicable)
	@@	(not applicable)

Note: Always specify pointer controls before the variable for which they apply. △

See for more information on how SAS determines the pointer position.

Using Line-hold Specifiers

Line-hold specifiers keep the pointer on the current output line when

- more than one PUT statement writes to the same output line
- a PUT statement writes values from more than one observation to the same output line.

Without line-hold specifiers, each PUT statement in a DATA step writes a new output line.

In the PUT statement, trailing @ and double trailing @@ produce the same effect. Unlike the INPUT statement, the PUT statement does not automatically release a line that is held by a trailing @ when the DATA step begins a new iteration. SAS releases the current output line that is held by a trailing @ or double trailing @ when it encounters:

- a PUT statement without a trailing @
- a PUT statement that uses _BLANKPAGE_ or _PAGE_
- the end of the current line (determined by the current value of the LRECL= or LINESIZE= option in the FILE statement, if specified, or the LINESIZE= system option)
- the end of the last iteration of the DATA step.

Using a trailing @ or double trailing @ can cause SAS to attempt to write past the current line length because the pointer value is unchanged when the next PUT statement executes. See .

Pointer Location after a Value Is Written

Understanding the location of the output pointer after a value is written is important, especially if you combine output styles in a single PUT statement. The pointer location after a value is written depends on which output style you use and whether a character

string or a variable is written. With column or formatted output, the pointer is set to the first column after the end of the field that is specified in the PUT statement. These two styles write only variable values.

With list output or named output, the pointer is positioned in the second column after a variable value because PUT skips a column automatically after each value is written. However, when a PUT statement uses list output to write a character string, the pointer is positioned in the first column after the string. If you do not use a line pointer control or column output after a character string is written, add a blank space to the end of the character string to separate it from the next value.

After an _INFILE_ specification, the pointer is located in the first column after the record written from the current input file.

When the output pointer is in the upper left corner of a page,

- PUT _BLANKPAGE_ writes a blank page and moves the pointer to the top of the next page.
- PUT _PAGE_ leaves the pointer in the same position.

You can determine the current location of the pointer by examining the variables that are specified with the COLUMN= option and the LINE= option in the FILE statement.

When the Pointer Goes Past the End of a Line

SAS does not write an output line that is longer than the current output line length. The line length of the current output file is determined by

- the value of the LINESIZE= option in the current FILE statement
- the value of the LINESIZE= system option (for print files)
- the LRECL= option in the current FILE statement (for nonprint files).

You may inadvertently position the pointer beyond the current line length with one or more of these specifications:

- a + pointer control with a value that moves the pointer to a column beyond the current line length
- a column range that exceeds the current line length (for example, PUT X 90 – 100 when the current line length is 80)
- a variable value or character string that does not fit in the space that remains on the current output line.

By default, when PUT attempts to write past the end of the current line, SAS withholds the entire item that overflows the current line, writes the current line, then writes the overflow item on a new line, starting in column 1. See the FLOWOVER, DROPOVER, and STOPOVER options in the statement “FILE” on page 802.

Arrays

You can use the PUT statement to write an array element. The subscript is any SAS expression that results in an integer when the PUT statement executes. You can use an array reference in a *numeric-variable* construction with a pointer control if you enclose the reference in parentheses, as shown here:

- @(*array-name*{i})
- +(*array-name*{i})
- #(*array-name*{i})

Use the array subscript asterisk (*) to write all elements of a previously defined array to a file. SAS allows single- or multidimensional arrays, but it does not permit a _TEMPORARY_ array. Enclose the subscript in braces, brackets, or parentheses, and

print the array using list, column, formatted, or named output. With list output, the form of this statement is

```
PUT array-name{*};
```

With formatted output, the form of this statement is

```
PUT array-name{*}(format|format.list)
```

The format in parentheses follows the array reference.

Comparisons

- The PUT statement writes variable values and character strings to the SAS log or to an external file while the INPUT statement reads raw data in external files or data lines entered instream.
- Both the INPUT and the PUT statements use the trailing @ and double trailing @ line-hold specifiers to hold the current line in the input or output buffer, respectively. In an INPUT statement, a double trailing @ holds a line in the input buffer from one iteration of the DATA step to the next. In a PUT statement, however, a trailing @ has the same effect as a double trailing @; both hold a line across iterations of the DATA step.
- Both the PUT and OUTPUT statements create output in a DATA step. The PUT statement uses an output buffer and writes output lines to a file, the SAS log, or your display. The OUTPUT statement uses the program data vector and writes observations to a SAS data set.

Examples

Example 1: Using Multiple Output Styles in One PUT Statement This example uses several output styles in a single PUT statement:

```
options yearcutoff= 1920;

data club1;
   input idno name $ startwght date : date7.;
   put name 'on ' date mmddyy8. ' weighs '
       startwght +(-1) '.' idno= 32-40;
   datalines;
032 David 180 25nov99
049 Amelia 145 25nov99
219 Alan 210 12nov99
;
```

The types of output styles are

The values for ...	Are written with ...
NAME, STARTWGHT	list output
DATE	formatted output
IDNO	named output

The PUT statement also uses pointer controls and specifies both character strings and variable names.

The program writes the following lines to the SAS log:*

```
----+----1----+----2----+----3----+----4
David on 11/25/99 weighs 180.  idno=1032
Amelia on 11/25/99 weighs 145. idno=1049
Alan on 11/12/99 weighs 210.   idno=1219
```

Blank spaces are inserted at the start and the end of the character strings to change the pointer position. These spaces separate the value of a variable from the character string. The +(-1) pointer control moves the pointer backward to remove the unwanted blank that occurs between the value of STARTWGHT and the period. For more information on how to position the pointer, see .

Example 2: Moving the Pointer within a Page These PUT statements show how to use column and line pointer controls to position the output pointer.

- To move the pointer to a specific column, use @ followed by the column number, variable, or expression whose value is that column number. For example, this statement moves the pointer to column 15 and writes the value of TOTAL SALES using list output:

  ```
  put @15 totalsales;
  ```

  This PUT statement moves the pointer to the value that is specified in COLUMN and writes the value of TOTALSALES with the COMMA6 format:

  ```
  data _null_;
     set carsales;
     column=15;
     put @column totalsales comma6.;
  run;
  ```

- This program shows two techniques to move the pointer backward:

  ```
  data carsales;
     input item $10. jan : comma5.
           feb : comma5. mar : comma5.;
     saleqtr1=sum(jan,feb,mar);
  /* an expression moves pointer backward */
     put '1st qtr sales for ' item
         'is ' saleqtr1 : comma6. +(-1) '.';
  /* a numeric variable with a negative
      value moves pointer backward.      */
     x=-1;
     put '1st qtr sales for ' item
         'is ' saleqtr1 : comma5. +x '.';
     datalines;
  trucks         1,382     2,789     3,556
  vans           1,265     2,543     3,987
  sedans         2,391     3,011     3,658
  ;
  ```

  Because the value of SALEQTR1 is written with modified list output, the pointer moves automatically two spaces. For more information, see "How Modified List Output and Formatted Output Differ" on page 985. To remove the unwanted blank that occurs between the value and the period, move the pointer backward by one space.

* The ruled line is for illustrative purposes only; the PUT statement does not generate it.

The program writes the following lines to the SAS log:

```
----+----1----+----2----+----3----+----4
st qtr sales for trucks is 7,727.
st qtr sales for trucks is 7,727.
st qtr sales for vans is 7,795.
st qtr sales for vans is 7,795.
st qtr sales for sedans is 9,060.
st qtr sales for sedans is 9,060.
```

- This program uses a PUT statement with the / line pointer control to advance to the next output line:

```
data _null_;
   set carsales end=lastrec;
   totalsales+saleqtr1;
   if lastrec then
     put @2 'Total Sales for 1st Qtr'
         / totalsales 10-15;
run;
```

After the DATA steps calculates TOTALSALES using all the observations in the CARSALES data set, the PUT statement executes. It writes a character string beginning in column 2 and moves the next line to write the value of TOTALSALES in columns 10 through 15:

```
----+----1----+----2----+----3
 Total Sales for 1st Qtr
          24582
```

Example 3: Moving the Pointer to a New Page This example creates a data set called STATEPOP, which contains information from the 1990 U.S. census about the population in metropolitan and nonmetropolitan areas. It executes the FORMAT procedure to group the 50 states and the District of Columbia into four regions. It then uses the IF and the PUT statements to control the printed output.

```
options pagesize=24 linesize=64 nodate pageno=1;

title1;

data statepop;
   input state $ cityp90 ncityp90 region @@;
   label cityp90= '1990 metropolitan population
                       (million)'
         ncityp90='1990 nonmetropolitan population
                       (million)'
         region=  'Geographic region';
   datalines;
ME    .443    .785  1   NH    .659    .450  1
VT    .152    .411  1   MA   5.788    .229  1
RI    .938    .065  1   CT   3.148    .140  1
NY  16.515   1.475  1   NJ   7.730     .A   1
PA  10.083   1.799  1   DE    .553    .113  2
MD   4.439    .343  2   DC    .607      .   2
VA   4.773   1.414  2   WV    .748   1.045  2
NC   4.376   2.253  2   SC   2.423   1.064  2
GA   4.352   2.127  2   FL  12.023    .915  2
KY   1.780   1.906  2   TN   3.298   1.579  2
```

```
AL   2.710   1.331  2   MS    .776   1.798  2
AR   1.040   1.311  2   LA   3.160   1.060  2
OK   1.870   1.276  2   TX  14.166   2.821  2
OH   8.826   2.021  3   IN   3.962   1.582  3
IL   9.574   1.857  3   MI   7.698   1.598  3
WI   3.331   1.561  3   MN   3.011   1.364  3
IA   1.200   1.577  3   MO   3.491   1.626  3
ND    .257    .381  3   SD    .221    .475  3
NE    .787    .791  3   KS   1.333   1.145  3
MT    .191    .608  4   ID    .296    .711  4
WY    .134    .319  4   CO   2.686    .608  4
NM    .842    .673  4   AZ   3.106    .559  4
UT   1.336    .387  4   NV   1.014    .183  4
WA   4.036    .830  4   OR   1.985    .858  4
CA  28.799    .961  4   AK    .226    .324  4
HI    .836    .272  4
;

proc format;
   value regfmt 1='Northeast'
                2='South'
                3='Midwest'
                4='West';
run;

data _null_;
   set statepop;
   by region;
   pop90=sum(cityp90,ncityp90);
   file print;
   put state 1-2 @5 pop90 7.3 ' million';
   if first.region then
      regioncitypop=0;      /* new region */
   regioncitypop+cityp90;
   if last.region then
      do;
         put // '1990 US CENSUS for ' region regfmt.
              / 'Total Urban Population: '
                 regioncitypop' million' _page_;
      end;
run;
```

Output 6.10 PUT Statement Output for the Northeast Region

```
                                                                        1
ME    1.228 million
NH    1.109 million
VT    0.563 million
MA    6.017 million
RI    1.003 million
CT    3.288 million
NY   17.990 million
NJ    7.730 million
PA   11.882 million


1990 US CENSUS for Northeast
Total Urban Population: 45.456  million
```

PUT _PAGE_ advances the pointer to line 1 of the new page when the value of LAST.REGION is 1. The example prints a footer message before exiting the page.

Example 4: Underlining Text This example uses OVERPRINT to underscore a value written by a previous PUT statement:

```
data _null_;
   input idno name $ startwght;
   file file-specification print;
   put name 1-10 @15 startwght 3.;
   if startwght > 200 then
     put overprint @15 '___';
   datalines;
032 David 180
049 Amelia 145
219 Alan 210
;
```

The second PUT statement underlines weights above 200 on the output line the first PUT statement prints.

This PUT statement uses OVERPRINT with both a column pointer control and a line pointer control:

```
   put @5 name $8. overprint @5 8*'_'
     / @20 address;
```

The PUT statement writes a NAME value, underlines it by overprinting eight underscores, and moves the output pointer to the next line to write an ADDRESS value.

Example 5: Holding and Releasing Output Lines This DATA step demonstrates how to hold and release an output line with a PUT statement:

```
data _null_;
   input idno name $ startwght 3.;
   put name @;
   if startwght ne . then
     put @15 startwght;
   else put;
   datalines;
032 David 180
049 Amelia 145
126 Monica
```

```
219 Alan 210
;
```

In this example,

- the trailing @ in the first PUT statement holds the current output line after the value of NAME is written
- if the condition is met in the IF-THEN statement, the second PUT statement writes the value of STARTWGHT and releases the current output line
- if the condition is not met, the second PUT never executes. Instead, the ELSE PUT statement executes. This releases the output line and positions the output pointer at column 1 in the output buffer.

The program writes the following lines to the SAS log:

```
----+----1----+----2
David          180
Amelia         145
Monica
Alan           210
```

Example 6: Writing the Current Input Record to the Log When a value for ID is less than 1000, PUT _INFILE_ executes and writes the current input record to the SAS log. The DELETE statement prevents the DATA step from writing the observation to the TEAM data set.

```
data team;
   input id team $ score1 score2;
   if id le 1000 then
     do;
       put _infile_;
       delete;
     end;
   datalines;
032 red 180 165
049 yellow 145 124
219 red 210 192
;
```

The program writes the following lines to the SAS log:

```
----+----1----+----2
219 red 210 192
```

See Also

Statements:

"FILE" on page 802
"PUT, Column" on page 977
"PUT, Formatted" on page 979
"PUT, List" on page 983
"PUT, Named" on page 987
"PUT, _ODS_" on page 989

System Options:

"LINESIZE=" on page 1112
"PAGESIZE=" on page 1133

PUT, Column

Writes variable values in the specified columns in the output line

Valid: in a DATA step
Category: File-handling
Type: Executable

Syntax

PUT *variable start-column* <— *end-column*>
<*.decimal–places*> <@ | @@>;

Arguments

variable
names the variable whose value is written.

start-column
specifies the first column of the field where the value is written in the output line.

— *end-column*
specifies the last column of the field for the value.

Tip: If the value occupies only one column in the output line, omit *end-column*.

Example: Because *end-column* is omitted, the values for the character variable GENDER occupy only column 16:

```
put name 1-10 gender 16;
```

.decimal-places
specifies the number of digits to the right of the decimal point in a numeric value.

Range: positive integer

Tip: If you specify 0 for *d* or omit *d*, the value is written without a decimal point.

Featured in: "Examples" on page 978

@| @@
holds an output line for the execution of the next PUT statement even across iterations of the DATA step. These line-hold specifiers are called *trailing @* and *double trailing @*.

Requirement: The trailing @ or double trailing @ must be the last item in the PUT statement.

See: "Using Line-hold Specifiers" on page 969

Details

With column output, the column numbers indicate the position that each variable value will occupy in the output line. If a value requires fewer columns than specified, a character variable is left-aligned in the specified columns, and a numeric variable is right-aligned in the specified columns.

There is no limit to the number of column specifications you can make in a single PUT statement. You can write anywhere in the output line, even if a value overwrites columns that were written earlier in the same statement. You can combine column output with any of the other output styles in a single PUT statement. For more information, see .

Examples

Use column output in the PUT statement as shown here.

□ This PUT statement uses column output:

```
data _null_;
   input name $ 1-18 score1 score2 score3;
   put name 1-20 score1 23-25 score2 28-30
       score3 33-35;
   datalines;
Joseph                  11   32   76
Mitchel                 13   29   82
Sue Ellen               14   27   74
;
```

The program writes the following lines to the SAS log:*

```
----+----1----+----2----+----3----+----4
Joseph                 11   32   76
Mitchel                13   29   82
Sue Ellen              14   27   74
```

The values for the character variable NAME begin in column 1, the left boundary of the specified field (columns 1 through 20). The values for the numeric variables SCORE1 through SCORE3 appear flush with the right boundary of their field.

□ This statement produces the same output lines, but writes the SCORE1 value first and the NAME value last:

```
put score1 23-25 score2 28-30
    score3 33-35 name $ 1-20;
```

□ This DATA step specifies decimal points with column output:

* The ruled line is for illustrative purposes only; the PUT statement does not generate it.

```
data _null_;
    x=11;
    y=15;
    put x 10-18 .1 y 20-28 .1;
run;
```

This program writes the following line to the SAS log:

```
----+----1----+----2----+----3----+----4
              11.0      15.0
```

See Also

Statement:

"PUT" on page 962

PUT, Formatted

Writes variable values with the specified format in the output line

Valid: in a DATA step

Category: File-handling

Type: Executable

Syntax

PUT <*pointer-control*> *variable format.* <@ | @@>;

PUT <*pointer-control*> (*variable-list*) (*format-list*)
<@ | @@>;

Arguments

pointer-control
moves the output pointer to a specified line or column.

See: "Column Pointer Controls" on page 964 and "Line Pointer Controls" on page 966

Featured in: Example 1 on page 982

variable
names the variable whose value is written.

(*variable-list*)
specifies a list of variables whose values are written.

Requirement: The (*format-list*) must follow the (*variable-list*).

See: "How to Group Variables and Formats" on page 981

Featured in: Example 1 on page 982

format.
specifies a format to use when the variable values are written. To override the default alignment, you can add an alignment specification to a format:

-L left aligns the value.

-C centers the value.

-R right aligns the value.

Tip: Ensure that the format width provides enough space to write the value and any commas, dollar signs, decimal points, or other special characters that the format includes.

Example: This PUT statement uses the format dollar7.2 to write the value of X:

```
put x dollar7.2;
```

When X is 100, the formatted value uses seven columns:

```
$100.00
```

Featured in: Example 2 on page 982

(*format-list*)
specifies a list of formats to use when the values of the preceding list of variables are written. In a PUT statement, a *format-list* can include

format.
specifies the format to use to write the variable values.

Tip: You can specify either a SAS format or a user-written format. See Chapter 3, "Formats," on page 49.

pointer-control
specifies one of these pointer controls to use to position a value: @, #, /, +, and OVERPRINT.

Example: Example 1 on page 982

character-string
specifies one or more characters to place between formatted values.

Example: This statement places a hyphen between the formatted values of CODE1, CODE2, and CODE3:

```
put bldg $ (code1 code2 code3) (3. '-');
```

See: Example 1 on page 982

*n**
specifies to repeat *n* times the next format in a format list.

Example: This statement uses the 7.2 format to write GRADES1, GRADES2, and GRADES3 and the 5.2 format to write GRADES4 and GRADES5:

```
put (grades1-grades5) (3*7.2, 2*5.2);
```

Restriction: The (*format-list*) must follow (*variable-list*).

See Also: "How to Group Variables and Formats" on page 981

@| @@
holds an output line for the execution of the next PUT statement even across iterations of the DATA step. These line-hold specifiers are called *trailing* @ and *double trailing* @.

Restriction: The trailing @ or double trailing @ must be the last item in the PUT statement.

See: "Using Line-hold Specifiers" on page 969

Details

Using Formatted Output The Formatted output describes the output lines by listing the variable names and the formats to use to write the values. You can use a SAS format or a user-written format to control how SAS prints the variable values. For a complete description of the SAS formats, see “Definition” on page 51 .

With formatted output, the PUT statement uses the format that follows the variable name to write each value. SAS does not automatically add blanks between values. If the value uses fewer columns than specified, character values are left-aligned and numeric values are right-aligned in the field that is specified by the format width.

Formatted output, combined with pointer controls, makes it possible to specify the exact line and column location to write each variable. For example, this PUT statement uses the dollar7.2 format and centers the value of X starting at column 12:

```
put @12 x dollar7.2-c;
```

How to Group Variables and Formats When you want to write values in a pattern on the output lines, use format lists to shorten your coding time. A format list consists of the corresponding formats separated by either blanks or commas and enclosed in parentheses. It must follow the names of the variables enclosed in parentheses.

For example, this statement uses a format list to write the five variables SCORE1 through SCORE5, one after another, using four columns for each value with no blanks in between:

```
put (score1-score5) (4. 4. 4. 4. 4.);
```

A shorter version of the previous statement is

```
put (score1-score5) (4.);
```

You can include any of the pointer controls (@, #, /, +, and OVERPRINT) in the list of formats, as well as *n**, and a character string. You can use as many format lists as necessary in a PUT statement, but do not nest the format lists. After all the values in the variable list are written, the PUT statement ignores any directions that remain in the format list.

For example, this format list includes more specifications than are necessary when the PUT statement writes the last variable:

```
data _null_;
   input x y z;
   put (x y z) (2.,+1);
   datalines;
2 24 36
0 20 30
;
```

The PUT statement writes the value of X using the 2. format. Then, the +1 column pointer control moves the pointer forward one column. Next, the value of Y is written with the 2. format. Again, the +1 column pointer moves the pointer forward one column. Then, the value of Z is written with the 2. format. For the third iteration, the PUT statement ignores the +1 pointer control.

These lines are written to the SAS log:*

---

* The ruled line is for illustrative purposes only; the PUT statement does not generate it.

```
----+----1----+
2 24 36
0 20 30
```

You can also specify a reference to all elements in an array as (*array-name* {*}), followed by a list of formats. You cannot, however, specify the elements in a _TEMPORARY_ array in this way. This PUT statement specifies an array name and a format list:

```
put (array1{*}) (4.);
```

For more information on how to reference an array, see .

Examples

Example 1: Writing a Character between Formatted Values This example formats some values and writes a - (hyphen) between the values of variables BLDG and ROOM:

```
data _null_;
   input name & $15. bldg $ room;
   put name @20 (bldg room) ($1. "-" 3.);
   datalines;
Bill Perkins  J 126
Sydney Riley  C 219
;
```

These lines are written to the SAS log:

```
Bill Perkins       J-126
Sydney Riley       C-219
```

Example 2: Overriding the Default Alignment of Formatted Values This example includes an alignment specification in the format:

```
data _null_;
   input name $ 1-12 score1 score2 score3;
   put name $12.-r +3 score1 3. score2 3.
       score3 4.;
   datalines;
Joseph                  11   32   76
Mitchel                 13   29   82
Sue Ellen               14   27   74
;
```

These lines are written to the log:

```
----+----1----+----2----+----3----+----4
      Joseph    11 32  76
     Mitchel    13 29  82
   Sue Ellen    14 27  74
```

The value of the character variable NAME is right-aligned in the formatted field. (Left alignment is the default for character variables.)

See Also

Statement:

"PUT" on page 962

PUT, List

Writes variable values and the specified character strings in the output line

Valid: in a DATA step

Category: File-handling

Type: Executable

Syntax

PUT <*pointer-control*> *variable* <@ | @@>;

PUT <*pointer-control*> <*n**>'*character-string*' <@ | @@>;

PUT <*pointer-control*> *variable* : *format*.<@ | @@>;

Arguments

pointer-control

moves the output pointer to a specified line or column.

See: "Column Pointer Controls" on page 964 and "Line Pointer Controls" on page 966

Featured in: Example 2 on page 986

variable

names the variable whose value is written.

Featured in: Example 1 on page 985

n*

specifies to repeat *n* times the subsequent character string.

Example: This statement writes a line of 132 underscores:

```
put 132*'_';
```

'***character-string***'

specifies a string of text, enclosed in quotation marks, to write.

Interaction: When insufficient space remains on the current line to write the entire text string, SAS withholds the entire string and writes the current line. Then it writes the text string on a new line, starting in column 1. For more information, see .

Tip: To avoid misinterpretation, always put a space after a closing quotation mark in a PUT statement.

Tip: If you follow a quotation mark with X, SAS interprets the text string as a hexadecimal constant.

See Also: "How List Output is Spaced" on page 984

Featured in: Example 2 on page 986

:
allows you to specifiy a format that the PUT statement uses to write the variable value. All leading and trailing blanks are still deleted, and each value is followed by a single blank.

Requirement: You must specify a format.

See: "How Modified List Output and Formatted Output Differ" on page 985

Featured in: Example 3 on page 986

format.
specifies a format to use when the data values are written.

Tip: You can specify either a SAS format or a user-written format. See Chapter 3, "Formats," on page 49.

Featured in: Example 3 on page 986

@ | @@
holds an output line for the execution of the next PUT statement even across iterations of the DATA step. These line-hold specifiers are called *trailing* @ and *double trailing* @.

Restriction: The trailing @ or double-trailing @ must be the last item in the PUT statement.

See: "Using Line-hold Specifiers" on page 969

Details

Using List Output With list output, you list the names of the variables whose values you want written, or you specify a character string in quotation marks. The PUT statement writes a variable value, inserts a single blank, and then writes the next value. Missing values for numeric variables are written as a single period. Character values are left-aligned in the field; leading and trailing blanks are removed. To include blanks (in addition to the blank inserted after each value), use formatted or column output instead of list output.

There are two types of list output:

- simple list output
- modified list output.

Modified list output increases the versatility of the PUT statement because you can specify a format to control how the variable values are written. See Example 3 on page 986.

How List Output is Spaced List output uses different spacing methods when it writes variable values and character strings. When a variable is written with list output, SAS automatically inserts a blank space. The output pointer stops at the second column that follows the variable value. However, when a character string is written, SAS does not automatically insert a blank space. The output pointer stops at the column that immediately follows the last character in the string.

To avoid spacing problems when both character strings and variable values are written, you may want to use a blank space as the last character in a character string. When a character string that provides punctuation follows a variable value, you need to

move the output pointer backward. This prevents an unwanted space from appearing in the output line. See Example 2 on page 986.

Comparisons

How Modified List Output and Formatted Output Differ List output and formatted output use different methods to determine how far to move the pointer after a variable value is written. Therefore, modified list output, which uses formats, and formatted output produce different results in the output lines. Modified list output writes the value, inserts a blank space, and moves the pointer to the next column. Formatted output moves the pointer the length of the format, even if the value does not fill that length. The pointer moves to the next column; an intervening blank is not inserted.

The following DATA step uses modified list output to write each output line:

```
data _null_;
   input x y;
   put x : comma10.2 y : 7.2;
   datalines;
2353.20 7.10
6231 121
;
```

These lines are written to the SAS log:

```
----+----1----+----2
2,353.20 7.10
6,231.00 121.00
```

In comparison, the following example uses formatted output:

```
put x comma10.2 y 7.2;
```

These lines are written to the SAS log, with the values aligned in columns:

```
----+----1----+----2
 12,353.20   7.10
  6,231.00 121.00
```

Examples

Example 1: Writing Values with List Output

This DATA step uses a PUT statement with list output to write variable values to the SAS log:

```
data _null_;
   input name $ 1-10 sex $ 12 age 15-16;
   put name sex age;
   datalines;
Joseph     M  13
Mitchel    M  14
Sue Ellen  F  11
;
```

These lines are written to the log:

```
----+----1----+----2----+----3----+----4
Joseph M 13
Mitchel M 14
Sue Ellen F 11
```

By default, the values of the character variable NAME are left-aligned in the field.

Example 2: Writing Character Strings and Variable Values This PUT statement adds a space to the end of a character string and moves the output pointer backward to prevent an unwanted space from appearing in the output line after the variable STARTWGHT:

```
data _null_;
   input idno name $ startwght;
   put name 'weighs ' startwght +(-1) '.';
   datalines;
032 David 180
049 Amelia 145
219 Alan 210
;
```

These lines are written to the SAS log:

```
David weighs 180.
Amelia weighs 145.
Alan weighs 210.
```

The blank space at the end of the character string changes the pointer position. This space separates the character string from the value of the variable that follows. The +(-1) pointer control moves the pointer backward to remove the unwanted blank that occurs between the value of STARTWGHT and the period.

Example 3: Writing Values with Modified List Output This DATA step uses modified list output to write several variable values in the output line:

```
data _null_;
   input salesrep : $10. tot : comma6.
         date : date9.;
   put 'Week of ' date : worddate15.
       salesrep : $12. 'sales were '
       tot : dollar9.  + (-1) '.';
   datalines;
Wilson 15,300 12OCT1999
Hoffman 9,600 12OCT1999
;
```

These lines appear in the SAS log:

```
Week of Oct 12, 1999 Wilson sales were $15,300.
Week of Oct 12, 1999 Hoffman sales were $9,600.
```

See Also

Statements:

"PUT" on page 962

"PUT, Formatted" on page 979

PUT, Named

Writes variable values after the variable name and an equal sign

Valid: in a DATA step

Category: File-handling

Type: Executable

Syntax

PUT <*pointer-control*> *variable*= <*format.*> <@ | @@>;

PUT *variable*= *start-column* <— *end-column*>
<*.decimal-places*> <@ | @@>;

Arguments

pointer-control
moves the output pointer to a specified line or column in the output buffer.

See: "Column Pointer Controls" on page 964 and "Line Pointer Controls" on page 966

variable=
names the variable whose value is written by the PUT statement in the form

```
variable=value
```

format.
specifies a format to use when the variable values are written.

Tip: Ensure that the format width provides enough space to write the value and any commas, dollar signs, decimal points, or other special characters that the format includes.

Example: This PUT statement uses the format DOLLAR7.2 to write the value of X:

```
put x= dollar7.2;
```

When X=100, the formatted value uses seven columns:

```
X=$100.00
```

See: "Formatting Named Output" on page 988

start-column
specifies the first column of the field where the variable name, equal sign, and value are to be written in the output line.

— end-column
determines the last column of the field for the value.

Tip: If the variable name, equal sign, and value require more space than the columns specified, PUT will write past the end column rather than truncate the value. You must leave enough space before beginning the next value.

.decimal-places
specifies the number of digits to the right of the decimal point in a numeric value. If you specify 0 for *d* or omit *d*, the value is written without a decimal point.

Range: positive integer

@ | @@
holds an output line for the execution of the next PUT statement even across iterations of the DATA step. These line-hold specifiers are called *trailing @* and *double trailing @*.

Restriction: The trailing @ or double trailing @ must be the last item in the PUT statement.

See: "Using Line-hold Specifiers" on page 969

Details

Using Named Output With named output, follow the variable name with an equal sign in the PUT statement. You can use either list output, column output, or formatted output specifications to indicate how to position the variable name and values. To insert a blank space between each variable value automatically, use list output. To align the output in columns, use pointer controls or column specifications.

Formatting Named Output You can specify either a SAS format or a user-written format to control how SAS prints the variable values. The width of the format does *not* include the columns required by the variable name and equal sign. To align a formatted value, SAS deletes leading blanks and writes the variable value immediately after the equal sign. SAS does not align on the right side of the formatted length, as in unnamed formatted output.

For a complete description of the SAS formats, see “Definition” on page 51 .

Examples

Use named output in the PUT statement as shown here.

□ This PUT combines named output with column pointer controls to align the output:

```
data _null_;
   input name $ 1-18 score1 score2 score3;
   put name = @20 score1= score3= ;
   datalines;
Joseph                  11   32   76
Mitchel                 13   29   82
Sue Ellen               14   27   74
;
```

The program writes the following lines to the SAS log:

```
----+----1----+----2----+----3----+----4
NAME=Joseph         SCORE1=11 SCORE3=76
NAME=Mitchel        SCORE1=13 SCORE3=82
NAME=Sue Ellen      SCORE1=14 SCORE3=74
```

□ This example specifies an output format for the variable AMOUNT:

```
put item= @25 amount= dollar12.2;
```

When the value of ITEM is binders and the value of AMOUNT is 153.25, this output line is produced:

```
----+----1----+----2----+----3----+----4
ITEM=binders            AMOUNT=$153.25
```

See Also

Statement:

"PUT" on page 962

PUT, _ODS_

Writes data values to a special buffer from which they can be written to the data component, and formatted by ODS destinations

Valid: in a DATA step

Category: File-handling

Type: Executable

Syntax

PUT <*specification(s)*><_ODS_><@|@@> ;

Arguments

specification
specifies the variables to write and where to write them. Each specification has the following form:
<ods-pointer-control>variable

ods-pointer-control

variable
identifies the variable to write.

Options

ODS
moves data values for all columns to a buffer. The order of these columns is determined by the orderthat is specified by the COLUMNS= or VARIABLES= suboption in the ODS option in the FILE statement. If you do not specify either of these options, the order of the variables in the program data vector determines their order in the buffer.

The PUT statement writes this buffer to the data component.

Interaction: You can use _ODS_ in a PUT statement that specifies the placement of individual variables. _ODS_ writes to a particular column only if a PUT

statement has not already written a variable to that column. The position of _ODS_ in the PUT statement does not affect the outcome in the data component, but it may affect performance.

Restriction: Use _ODS_ only if you have previously specified the ODS option in the FILE statement.

Tip: The order of the column in the data component matches the order of the columns in buffer. However, the template that is combined with the data component to produce the output object may override this order.

@ | @@
holds an output line for the execution of the next PUT statement across iterations of the DATA step. The line-hold specifiers are called *trailing @* and *double trailing @*.

Default: If you do not use @ or @@, each PUT statement in a DATA step writes a new line to the buffer.

Details

Within the DATA step, the ODS option in the FILE statement and the _ODS_ option in the PUT statement provide connections with the Output Delivery System (ODS). You use both of these connections to route the results of a DATA step to ODS. By default, when the DATA step uses ODS, ODS writes output objects to the procedure output and places links to them in the Results folder. You can use global ODS statements to write to other ODS destinations.

The FILE and PUT statements interact in the following ways:

- The ODS option in the FILE statement defines the structure of the data component that holds the results of the DATA step.
- The _ODS_ option in the PUT statement writes values (as specified by the ODS option in the FILE statement) into a special buffer. This buffer is written to the data component.
- The ODS option in the FILE statement binds the data component to a template to produce an output object. ODS sends this object to all open ODS destinations, each of which formats the object appropriately.

The ODS destinations are controlled by the global ODS statements. You can use an existing template or create your own with the TEMPLATE. procedure.

For details on using the Output Delivery System, see *The Complete Guide to the SAS Output Delivery System*.

Column Pointer Controls and ODS Column pointer controls in a DATA step that uses ODS differ slightly from column pointer controls in a DATA step that does not use ODS. ODS is not character-based. Therefore, in ODS a column contains the entire value of a variable. Column 1 contains the first variable in the output; column 2 contains the second variable, and so on.

Column pointer controls have the following general forms:

@ods-column

+ods-column

@ods-column
moves the pointer to the specified ODS column. *ods-column* can be a number, a numeric-variable, or an expression that identifies the column to write to.

Range: If *ods-column* is a number, it must be a positive integer.
If *ods-column* is a variable or an expression, SAS treats it as follows:

If the variable or expression is ...	SAS
not an integer	truncates the decimal portion and uses only the integer value
0 or negative	moves the pointer to column 1

Tip: By default, if *ods-column* exceeds the number of columns in the data component, ODS writes the current line, moves the pointer to the first column on the next line, and continues to process the PUT statement. You can alter this behavior with options in the FILE statement.

+ods-column
moves the pointer the specified number of columns. *ODS-column* can be a number, a numeric-variable, or an expression that specifies the number of columns to move the pointer.

Range: If *ods-column* is a number, it must be an integer. If *ods-column* is a variable or an expression, it does not have to be an integer. If it isn't, SAS truncates the decimal portion and uses the only the integer value.

Tip: If *ods-column* is greater than 0, the pointer moves to the right. If *ods-column* is less than 0, the pointer moves to the left. If *ods-column* is equal to 0, the pointer does not move.

If the current column position becomes less than 1, the pointer moves to column 1. If the current column position exceeds the number of columns in the data component, the ODS writes the current line, moves the pointer to the first column on the next line, and continues to process the PUT statement.

Line Pointer Controls Line pointer controls in a DATA step that uses ODS are the same as line pointer controls in a DATA step that does not use ODS. Line pointer controls have the following general forms:

#line

/

#line
moves the pointer to the specified line. *line* can be a number, a numeric-variable, or an expression that identifies the line to write to.

Range: If *line* is a number, it must be an integer. If *line* is a variable or an expression, it does not have to be an integer. If it isn't, SAS truncates the decimal portion and uses the only the integer value.

/
moves the pointer to the first column of the next line.

When the Pointer Goes Past the End of a Line In a DATA step that uses the Output Delivery System, the number of columns that is specified by the COLUMNS= or VARIABLES= suboption to the ODS option in the FILE statement determines the number of columns in the buffer, and eventually, in the data component. If you do not specify either of these options, the number of the variables in the program data vector determines the number of columns.

Note: The template that is combined with the data component to produce the output object may change the number of columns that actually appear in the output object. △

Using pointer controls and trailing @ or double trailing @, you may inadvertently position the pointer beyond the last column. You control how SAS handles this situation with options in the FILE statement.

REDIRECT

Points to different input or output SAS data sets when you execute a stored program

Valid: in a DATA step

Category: Action

Type: Executable

Requirement: You must specify the PGM= option in the DATA statement.

Syntax

REDIRECT INPUT | OUTPUT *old-name-1* = *new-name-1*<. . . *old-name-n* = *new-name-n*>;

Arguments

INPUT | OUTPUT
specifies whether to redirect input or output data sets. When you specify INPUT, the REDIRECT statement associates the name of the input data set in the source program with the name of another SAS data set. When you specify OUTPUT, the REDIRECT statement associates the name of the output data set with the name of another SAS data set.

old-name
specifies the name of the input or output data set in the source program.

new-name
specifies the name of the input or output data set that you want SAS to process for the current execution.

Details

The REDIRECT statement is available only when you execute a stored program. For more information about stored programs, see the "Stored Compiled DATA Step Programs" chapter of *SAS Language Reference: Concepts*.

CAUTION:
Use care when you redirect input data sets. The number and attributes of variables in the input data sets that you read with the REDIRECT statement should match those of the input data sets in the MERGE, SET, MODIFY, or UPDATE statements of the source code. If the variable type attributes differ, the stored program stops processing and an appropriate error message is sent to the SAS log. If the variable length attributes differ, the length of the variable in the source code data set determines the length of the variable in the redirected data set. Extra variables in the redirected data sets do not cause the stored program to stop processing, but the results may not be what you expect. △

Comparison

The REDIRECT statement applies only to SAS data sets. To redirect input and output stored in external files, include a FILENAME statement to associate the fileref in the source program with different external files.

Examples

This example executes the stored program called STORED.SAMPLE. The REDIRECT statement specifies the source of the input data as BASE.SAMPLE. The output data set from this execution of the program is redirected and stored in a data set named SUMS.SAMPLE.

```
libname stored 'SAS-data-library';
libname base 'SAS-data-library';
libname sums 'SAS-data-library';

data pgm=stored.sample;
   redirect input in.sample=base.sample;
   redirect output out.sample=sums.sample;
run;
```

See Also

Statement:
"DATA" on page 774
"Stored Compiled DATA Step Programs" in *SAS Language Reference: Concepts*

REMOVE

Deletes an observation from a SAS data set

Valid: in a DATA step
Category: Action
Type: Executable
Restriction: Use only with a MODIFY statement.

Syntax

REMOVE <*data-set-name(s)*>;

Without Arguments

If you specify no argument, the REMOVE statement deletes the current observation from all data sets that are named in the DATA statement.

Arguments

data-set-name
specifies the data set in which the observation is deleted.

Restriction: The data set name must also appear in the DATA statement and in one or more MODIFY statements.

Details

The deletion of an observation can be physical or logical, depending on the engine that maintains the data set. Using REMOVE overrides the default replacement of observations. If a DATA step contains a REMOVE statement, you must explicitly program all output for the step.

Comparisons

- Using an OUTPUT, REPLACE, or REMOVE statement overrides the default write action at the end of a DATA step. (OUTPUT is the default action; REPLACE becomes the default action when a MODIFY statement is used.) If you use any of these statements in a DATA step, you must explicitly program all output for new observations.
- The OUTPUT, REPLACE, and REMOVE statements are independent of each other. More than one statement can apply to the same observation, as long as the sequence is logical.
- If both an OUTPUT and a REPLACE or REMOVE statement execute on a given observation, perform the OUTPUT action last to keep the position of the observation pointer correct.
- Because the REMOVE statement can perform a physical or a logical deletion, REMOVE is available with the MODIFY statement for all SAS data set engines. Both the DELETE and subsetting IF statements perform only physical deletions; therefore, they are not available with the MODIFY statement for certain engines.

Examples

This example removes one observation from a SAS data set.

```
libname perm 'SAS-data-library';


data perm.accounts;
   input AcctNumber Credit;
   datalines;
001 1500
002 4900
003 3000
;

data perm.accounts;
   modify perm.accounts;
   if AcctNumber=1002 then remove;
run;

proc print data=perm.accounts;
   title 'Edited Data Set';
run;
```

Here are the results of the PROC PRINT statement:

```
                    Edited Data Set                          1

                          Acct
                OBS     Number     Credit

                 1       1001       1500
                 3       1003       3000
```

See Also

Statements:

"DELETE" on page 783
"IF, Subsetting" on page 847
"MODIFY" on page 933
"OUTPUT" on page 958
"REPLACE" on page 997

RENAME

Specifies new names for variables in output SAS data sets

Valid: in a DATA step
Category: Information
Type: Declarative

Syntax

RENAME *old-name-1=new-name-1 . . . <old-name-n=new-name-n>*;

Arguments

old-name
specifies the name of a variable or variable list as it appears in the input data set, or in the current DATA step for newly created variables.

new-name
specifies the name or list to use in the output data set.

Details

The RENAME statement allows you to change the names of one or more variables, variables in a list, or a combination of variables and variable lists. The new variable names are written to the output data set only. Use the old variable names in programming statements for the current DATA step. RENAME applies to all output data sets.

Comparisons

- RENAME cannot be used in PROC steps, but the RENAME= data set option can.
- The RENAME= data set option allows you to specify the variables you want to rename for each input or output data set. Use it in input data sets to rename variables before processing.
- If you use the RENAME= data set option in an output data set, you must continue to use the old variable names in programming statements for the current DATA step. After your output data is created, you can use the new variable names.
- The RENAME= data set option in the SET statement renames variables in the input data set. You can use the new names in programming statements for the current DATA step.
- To rename variables as a file management task, use the DATASETS procedure or access the variables through the SAS windowing interface. These methods are simpler and do not require DATA step processing.

Examples

- These examples show the correct syntax for renaming variables using the RENAME statement:
  -
    ```
    rename street=address;
    ```
  -
    ```
    rename time1=temp1 time2=temp2 time3=temp3;
    ```
  -
    ```
    rename name=Firstname
           score1-score3=Newscore1-Newscore3;
    ```
- This example uses the old name of the variable in program statements. The variable Olddept is named Newdept in the output data set, and the variable Oldaccount is named Newaccount.

  ```
  rename Olddept=Newdept Oldaccount=Newaccount;
  if Oldaccount>5000;
  keep Olddept Oldaccount items volume;
  ```
- This example uses the old name OLDACCNT in the program statements. However, the new name NEWACCNT is used in the DATA statement because SAS applies the RENAME statement before it applies the KEEP= data set option.

  ```
  data market(keep=newdept newaccnt items
              volume);
     rename olddept=newdept
            oldaccnt=newaccnt;
     set sales;
     if oldaccnt>5000;
  run;
  ```
- The following example uses both a variable and a variable list to rename variables. New variable names appear in the output data set.

  ```
  data temp;
     input (score1-score3) (2.,+1) name $;
  ```

```
   rename name=Firstname
          score1-score3=Newscore1-Newscore3;
   datalines;
12 24 36 Lisa
22 44 66 Fran
;
```

See Also

Data Set Option:

"RENAME=" on page 35

REPLACE

Replaces an observation in the same location

Valid: in a DATA step

Category: Action

Type: Executable

Restriction: Use only with a MODIFY statement.

Syntax

REPLACE <*data-set-name-1*><. . .*data-set-name-n*>;

Without Arguments

If you specify no argument, the REPLACE statement writes the current observation to the same physical location from which it was read in all data sets that are named in the DATA statement.

Arguments

data-set-name
specifies the data set to which the observation is written.

Requirement: The data set name must also appear in the DATA statement and in one or more MODIFY statements.

Details

Using an explicit REPLACE statement overrides the default replacement of observations. If a DATA step contains a REPLACE statement, explicitly program all output for the step.

Comparisons

- Using an OUTPUT, REPLACE, or REMOVE statement overrides the default write action at the end of a DATA step. (OUTPUT is the default action; REPLACE becomes the default action when a MODIFY statement is used.) If you use any of these statements in a DATA step, you must explicitly program output of a new observation for the step.
- The OUTPUT, REPLACE, and REMOVE statements are independent of each other. More than one statement can apply to the same observation, as long as the sequence is logical.
- If both an OUTPUT and a REPLACE or REMOVE statement execute on a given observation, perform the OUTPUT action last to keep the position of the observation pointer correct.
- REPLACE writes the observation to the same physical location, while OUTPUT writes a new observation to the end of the data set.
- REPLACE can appear only in a DATA step that contains a MODIFY statement. You can use OUTPUT with or without MODIFY.

Examples

This example updates phone numbers in data set MASTER with values in data set TRANS. It also adds one new observation at the end of data set MASTER. The SYSRC autocall macro tests the value of _IORC_ for each attempted retrieval from MASTER. (SYSRC is part of the SAS autocall macro library.) The resulting SAS data set appears after the code:

```
data master;
   input FirstName $ id $ PhoneNumber;
   datalines;
Kevin ABCjkh 904
Sandi defsns 905
Terry ghitDP 951
Jason jklJWM 962
;

data trans;
   input FirstName $ id $ PhoneNumber;
   datalines;
. ABCjkh 2904
. defsns 2905
Madeline mnombt 2983
;

data master;
   modify master trans;
   by id;
      /* obs found in master  */
      /* change info, replace */
   if _iorc_ = %sysrc(_sok) then replace;

      /* obs not in master    */
   else if _iorc_ = %sysrc(_dsenmr) then
      do;
         /* reset _error_        */
```

```
         _error_=0;
         /* reset _iorc_         */
         _iorc_=0;
         /* output obs to master */
      output;
      end;
run;

proc print data=master;
   title 'MASTER with New Phone Numbers';
run;
```

```
                MASTER with New Phone Numbers                 3

                      First                    Phone
               OBS    Name          id        Number

                1     Kevin       ABCjkh       2904
                2     Sandi       defsns       2905
                3     Terry       ghitDP        951
                4     Jason       jklJWM        962
                5     Madeline    mnombt       2983
```

See Also

Statements:

"MODIFY" on page 933

"OUTPUT" on page 958

"REMOVE" on page 993

RETAIN

Causes a variable that is created by an INPUT or assignment statement to retain its value from one iteration of the DATA step to the next

Valid: in a DATA step

Category: Information

Type: Declarative

Syntax

RETAIN <*element-list(s)* <*initial-value(s)* | (*initial-value-1*) | (*initial-value-list-1*) > < . . . *element-list-n* <*initial-value-n* | (*initial-value-n*)| (*initial-value-list-n*)>>>;

Without Arguments

If you do not specify an argument, the RETAIN statement causes the values of all variables that are created with INPUT or assignment statements to be retained from one iteration of the DATA step to the next.

Arguments

element-list
specifies variable names, variable lists, or array names whose values you want retained.

Tip: If you specify _ALL_, _CHAR_, or _NUMERIC_, only the variables that are defined before the RETAIN statement are affected.

Tip: If a variable name is specified *only* in the RETAIN statement and you do not specify an initial value, the variable is *not* written to the data set, and a note stating that the variable is uninitialized is written to the SAS log. If you specify an initial value, the variable *is* written to the data set.

initial-value
specifies an initial value, numeric or character, for one or more of the preceding elements.

Tip: If you omit *initial-value*, the initial value is missing. *Initial-value* is assigned to all the elements that precede it in the list. All members of a variable list, therefore, are given the same initial value.

See Also: (*initial-value*) and (*initial-value-list*)

(*initial-value*)
specifies an initial value, numeric or character, for a single preceding element or for the first in a list of preceding elements.

(*initial-value-list*)
specifies an initial value, numeric or character, for individual elements in the preceding list. SAS matches the first value in the list with the first variable in the list of elements, the second value with the second variable, and so on.

Element values are enclosed in quotation marks. To specify one or more initial values directly, use the following format:

(*initial-value(s)*)

To specify an iteration factor and nested sublists for the initial values, use the following format:

<*constant-iter-value**> <(>*constant value* | *constant-sublist*<)>

Restriction: If you specify both an *initial-value-list* and an *element-list*, then *element-list* must be listed before *initial-value-list* in the RETAIN statement.

Tip: You can separate initial values by blank spaces or commas.

Tip: You can assign initial values to both variables and temporary data elements.

Tip: If there are more variables than initial values, the remaining variables are assigned an initial value of missing and SAS issues a warning message.

Details

Default DATA Step Behavior Without a RETAIN statement, SAS automatically sets variables that are assigned values by an INPUT or assignment statement to missing before each iteration of the DATA step.

Assigning Initial Values Use a RETAIN statement to specify initial values for individual variables, a list of variables, or members of an array. If a value appears in a

RETAIN statement, variables that appear before it in the list are set to that value initially. (If you assign different initial values to the same variable by naming it more than once in a RETAIN statement, SAS uses the last value.) You can also use RETAIN to assign an initial value other than the default value of 0 to a variable whose value is assigned by a sum statement.

Redundancy It is redundant to name any of these items in a RETAIN statement, because their values are automatically retained from one iteration of the DATA step to the next:

- variables that are read with a SET, MERGE, MODIFY or UPDATE statement
- a variable whose value is assigned in a sum statement
- the automatic variables _N_, _ERROR_, _I_, _CMD_, and _MSG_
- variables that are created by the END= or IN= option in the SET, MERGE, MODIFY, or UPDATE statement or by options that create variables in the FILE and INFILE statements
- data elements that are specified in a temporary array
- array elements that are initialized in the ARRAY statement
- elements of an array that have assigned initial values to any or all of the elements on the ARRAY statement.

You can, however, use a RETAIN statement to assign an initial value to any of the previous items, with the exception of _N_ and _ERROR_.

Comparisons

The RETAIN statement specifies variables whose values are *not set to missing* at the beginning of each iteration of the DATA step. The KEEP statement specifies variables that are to be included in any data set that is being created.

Examples

Example 1: Basic Usage

- This RETAIN statement retains the values of variables MONTH1 through MONTH5 from one iteration of the DATA step to the next:

```
retain month1-month5;
```

- This RETAIN statement retains the values of nine variables and sets their initial values:

```
retain month1-month5 1 year 0 a b c 'XYZ';
```

  The values of MONTH1 through MONTH5 are set initially to 1; YEAR is set to 0; variables A, B, and C are each set to the character value **XYZ**.

- This RETAIN statement assigns the initial value 1 to the variable MONTH1 only:

```
retain month1-month5 (1);
```

  Variables MONTH2 through MONTH5 are set to missing initially.

- This RETAIN statement retains the values of all variables that are defined earlier in the DATA step but not those defined *afterwards*:

```
retain _all_;
```

- Both of these statements assign initial values to VAR1 through VAR4:

- retain var1-var4 (1 2 3 4);
- retain var1-var4 (1,2,3,4);

Example 2: Overview of the RETAIN Operation This example shows how to use variable names and array names as elements in the RETAIN statement and shows assignment of initial values with and without parentheses:

```
data _null_;
   array City{3} $ City1-City3;
   array cp{3} Citypop1-Citypop3;
   retain Year Taxyear 1999 City ' '
          cp (10000,50000,100000);
   file file-specification print;
   put 'Values at beginning of DATA step:'
       / @3 _all_ /;
   input Gain;
   do i=1 to 3;
      cp{i}=cp{i}+Gain;
   end;
   put 'Values after adding Gain to city populations:'
       / @3 _all_;
   datalines;
5000
10000
;
```

The initial values assigned by RETAIN are as follows:

- Year and Taxyear are assigned the initial value 1999.
- City1, City2, and City3 are assigned missing values.
- Citypop1 is assigned the value 10000.
- Citypop2 is assigned 50000.
- Citypop3 is assigned 100000.

Here are the lines written by the PUT statements:

```
Values at beginning of DATA step:
  City1=  City2=  City3=  Citypop1=10000
  Citypop2=50000 Citypop3=100000
Year=1999 Taxyear=1999 Gain=. i=.
_ERROR_=0 _N_=1

Values after adding GAIN to city populations:
  City1=  City2=  City3=  Citypop1=15000
  Citypop2=55000 Citypop3=105000
Year=1999 Taxyear=1999 Gain=5000 i=4
_ERROR_=0 _N_=1
Values at beginning of DATA step:
  City1=  City2=  City3=  Citypop1=15000
  Citypop2=55000 Citypop3=105000
Year=1999 Taxyear=1999 Gain=. i=.
_ERROR_=0 _N_=2

Values after adding GAIN to city populations:
```

```
  City1=  City2=  City3=  Citypop1=25000
  Citypop2=65000 Citypop3=115000
Year=1999 Taxyear=1999 Gain=10000 i=4
_ERROR_=0 _N_=2
Values at beginning of DATA step:
  City1=  City2=  City3=  Citypop1=25000
  Citypop2=65000 Citypop3=115000
Year=1999 Taxyear=1999 Gain=. i=.
_ERROR_=0 _N_=3
```

The first PUT statement is executed three times, while the second PUT statement is executed only twice. The DATA step ceases execution when the INPUT statement executes for the third time and reaches the end of the file.

Example 3: Selecting One Value from a Series of Observations In this example, the data set ALLSCORES contains several observations for each identification number and variable ID. Different observations for a particular ID value may have different values of the variable GRADE. This example creates a new data set, CLASS.BESTSCORES, which contains one observation for each ID value. The observation must have the highest GRADE value of all observations for that ID in BESTSCORES.

```
libname class 'SAS-data-library';

proc sort data=class.allscores;
   by id;
run;

data class.bestscores;
   drop grade;
   set class.allscores;
   by id;
      /* Prevents HIGHEST from being reset*/
      /* to missing for each iteration.   */
   retain highest;
      /* Sets HIGHEST to missing for each */
      /* different ID value.              */
   if first.id then highest=.;
      /* Compares HIGHEST to GRADE in     */
      /* current iteration and resets     */
      /* value if GRADE is higher.        */
   highest=max(highest,grade);
   if last.id then output;
run;
```

See Also

Statements:

"Assignment" on page 761
"BY" on page 765
"INPUT" on page 876

RETURN

Stops executing statements at the current point in the DATA step and returns to a predetermined point in the step

Valid: in a DATA step
Category: Control
Type: Executable

Syntax

RETURN;

Without Arguments

The RETURN statement causes execution to stop at the current point in the DATA step, and returns control to a previous DATA step statement.

Details

The point to which SAS returns depends on the order in which statements are executed in the DATA step.

The RETURN statement is often used with the

- GO TO statement
- HEADER= option in the FILE statement
- LINK statement.

When RETURN causes a return to the beginning of the DATA step, an implicit OUTPUT statement writes the current observation to any new data sets (unless the DATA step contains an explicit OUTPUT statement, or REMOVE or REPLACE statements with MODIFY statements). Every DATA step has an implied RETURN as its last executable statement.

Examples

In this example, when the values of X and Y are the same, SAS executes the RETURN statement and adds the observation to the data set. When the values of X and Y are not equal, SAS executes the remaining statements and then adds the observation to the data set.

```
data survey;
   input x y;
   if x=y then return;
```

```
   put x= y=;
   datalines;
21 25
20 20
7 17
;
```

See Also

Statements:

"FILE" on page 802
"GO TO" on page 845
"LINK" on page 923

RUN

Executes the previously entered SAS statements

Valid: anywhere
Category: Program Control

Syntax

RUN <CANCEL>;

Without Arguments

Without arguments, the RUN statement executes the previously entered SAS statements.

Arguments

CANCEL
terminates the current step without executing it. SAS prints a message that indicates that the step was not executed.

CAUTION:
The CANCEL option does not prevent execution of a DATA step that contains a DATALINES or DATALINES4 statement. △

Details

Though the RUN statement is not required between steps in a SAS program, using it creates a step boundary and can make the SAS log easier to read.

Examples

- This RUN statement marks a step boundary and executes this PROC PRINT step:

```
proc print data=report;
   title 'Status Report';
run;
```

- This example shows the usefulness of the CANCEL option in a line prompt mode session. The fourth statement in the DATA step contains an invalid value for PI (4.13 instead of 3.14). RUN with CANCEL ends the DATA step and prevents it from executing.

```
data circle;
   infile file-specification;
   input radius;
   c=2*4.13*radius;
run cancel;
```

%RUN

Ends source statements following a %INCLUDE * statement

Valid: anywhere
Category: Program Control

Syntax

%RUN;

Without Arguments

The %RUN statement causes SAS to stop reading input from the terminal (including subsequent SAS statements on the same line as %RUN) and resume reading from the previous input source.

Details

Using the %INCLUDE statement with an asterisk specifies that you enter source lines from the keyboard.

Comparisons

The RUN statement executes previously entered DATA or PROC steps. The %RUN statement ends the prompting for source statements and returns program control to the original source program, when you use the %INCLUDE statement to allow data to be entered from the keyboard.

The type of prompt that you use depends on how you run the SAS session. The include operation is most useful in interactive line and noninteractive modes, but it can also be used in windowing and batch mode. When you are running SAS in batch mode, include the %RUN statement in the external file that is referenced by the SASTERM fileref.

Examples

- To request keyboard-entry source on a %INCLUDE statement, follow the statement with an asterisk:

```
%include *;
```

- When it executes this statement, SAS prompts you to enter source lines from the keyboard. When you finish entering code from the keyboard, type the following statement to return processing to the program that contains the %INCLUDE statement.

```
%run;
```

See Also

Statements:

"%INCLUDE" on page 851

"RUN" on page 1005

SELECT

Executes one of several statements or groups of statements

Valid: in a DATA step

Category: Control

Type: Executable

Syntax

SELECT <(*select-expression*)>;

WHEN-1 (*when-expression-1* <..., *when-expression-n*>) *statement*;

<... **WHEN**-*n* (*when-expression-1* <..., *when-expression-n*>) *statement*;>
<**OTHERWISE** *statement*;>

END;

Arguments

(select-expression)
specifies any SAS expression that evaluates to a single value.

(when-expression)
specifies any SAS expression, including a compound expression. SELECT requires you to specify at least one *when-expression*.

Tip: Separating multiple *when-expressions* with a comma is equivalent to separating them with the logical operator OR.

Tip: The way a *when-expression* is used depends on whether a *select-expression* is present.

Explanation: If the *select-expression* is present, SAS evaluates the *select-expression* and *when-expression*. SAS compares the two for equality and returns a value of true or false. If the comparison is true, *statement* is executed. If the comparison is false, execution proceeds either to the next *when-expression* in the current WHEN statement, or to the next WHEN statement if no more expressions are present. If no WHEN statements remain, execution proceeds to the OTHERWISE statement, if one is present. If the result of all SELECT-WHEN comparisons is false and no OTHERWISE statement is present, SAS issues an error message and stops executing the DATA step.

If no *select-expression* is present, the *when-expression* is evaluated to produce a result of true or false. If the result is true, *statement* is executed. If the result is false, SAS proceeds to the next *when-expression* in the current WHEN statement, or to the next WHEN statement if no more expressions are present, or to the OTHERWISE statement if one is present. (That is, SAS performs the action that is indicated in the first true WHEN statement.) If the result of all *when-expressions* is false and no OTHERWISE statement is present, SAS issues an error message. If more than one WHEN statement has a true *when-expression*, only the first WHEN statement is used; once a *when-expression* is true, no other *when-expressions* are evaluated.

statement
can be any executable SAS statement, including DO, SELECT, and null statements. You must specify the *statement* argument.

Details

The SELECT statement begins a SELECT group; SELECT groups contain WHEN statements that identify SAS statements that are executed when a particular condition is true. Use at least one WHEN statement in a SELECT group. An optional OTHERWISE statement specifies a statement to be executed if no WHEN condition is met. An END statement ends a SELECT group.

Null statements that are used in WHEN statements cause SAS to recognize a condition as true without taking further action. Null statements that are used in OTHERWISE statements prevent SAS from issuing an error message when all WHEN conditions are false.

Comparisons

- When you have a long series of mutually exclusive conditions, using a SELECT group is more efficient than using a series of IF-THEN statements because CPU time is reduced. Large numbers of conditions make a SELECT group more efficient than IF-THEN/ELSE statements because CPU time is reduced. SELECT groups also make the program easier to read and debug.
- Use IF-THEN/ELSE statements for programs with few statements. Subsetting IF statements should be used without a THEN clause to continue processing only those observations or records that meet the condition that is specified in the IF clause.

Examples

Example 1: Using Statements

```
select (a);
   when (1) x=x*10;
   when (2);
```

```
   when (3,4,5) x=x*100;
   otherwise;
end;
```

Example 2: Using DO Groups

```
select (payclass);
   when ('monthly') amt=salary;
   when ('hourly')
      do;
         amt=hrlywage*min(hrs,40);
         if hrs>40 then put 'CHECK TIMECARD';
      end;          /* end of do     */
   otherwise put 'PROBLEM OBSERVATION';
end;               /* end of select */
```

Example 3: Using a Compound Expression

```
select;
   when (mon in ('JUN', 'JUL', 'AUG')
   and temp>70) put 'SUMMER ' mon=;
   when (mon in ('MAR', 'APR', 'MAY'))
   put 'SPRING ' mon=;
   otherwise put 'FALL OR WINTER ' mon=;
end;
```

Example 4: Making Comparisons for Equality

```
   /* INCORRECT usage to select value of 2 */
select (x);
   /* evaluates T/F and compares for       */
   /* equality with x                      */
   when (x=2) put 'two';
end;


   /* correct usage */
select(x);
   /* compares 2 to x for equality */
   when (2) put 'two';
end;


   /* correct usage */
select;
   /* compares 2 to x for equality         */
   when (x=2) put 'two';
end;
```

See Also

Statements:

"DO" on page 789
"IF, Subsetting" on page 847
"IF-THEN/ELSE" on page 849

SET

Reads an observation from one or more SAS data sets

Valid: in a DATA step
Category: File-handling
Type: Executable

Syntax

SET< *SAS-data-set(s)* <(*data-set-options(s)*)>>
<*options*>;

Without Arguments

When you do not specify an argument, the SET statement reads an observation from the most recently created data set.

Arguments

SAS-data-set
specifies a one-level name, a two-level name, or one of the special SAS data set names.

See Also: See the "SAS Data Sets" chapter of *SAS Language Reference: Concepts* for a description of the levels of SAS data set names and when to use each level.

(*data-set-options*)
specifies actions SAS is to take when it reads variables or observations into the program data vector for processing.

See: Refer to Chapter 2, "Data Set Options," on page 5 for a list of the data set options to use with input data sets.

Options

END=*variable*
creates and names a temporary variable that contains an end-of-file indicator. The variable, which is initialized to zero, is set to 1 when SET reads the last observation of the last data set listed. This variable is not added to any new data set.

Restriction: END= cannot be used with POINT=. When random access is used, the END= variable is never set to 1.

Featured in: Example 11 on page 1016

KEY=*index*
provides nonsequential access to observations in a SAS data set, which are based on the value of an index variable or a key.

Range: Specify the name of a simple or a composite index of the data set that is being read.

Restriction: KEY= cannot be used with POINT=.

Tip: Using the _IORC_ automatic variable in conjunction with the SYSRC autocall macro provides you with more error-handling information than was previously available. When you use the SET statement with the KEY= option, the new automatic variable _IORC_ is created. This automatic variable is set to a return code that shows the status of the most recent I/O operation that is performed on an observation in a SAS data set. If the KEY= value is not found, the _IORC_ variable returns a value that corresponds to the SYSRC autocall macro's mnemonic _DSENOM and the automatic variable _ERROR_ is set to 1.

Featured in: Example 7 on page 1015 and Example 8 on page 1015.

See Also: For more information, see the description of the autocall macro SYSRC in *SAS Macro Language: Reference*.

See Also: UNIQUE option on page 1013

NOBS=*variable*
creates and names a temporary variable whose value is usually the total number of observations in the input data set or data sets. If more than one data set is listed in the SET statement, NOBS= the total number of observations in the data sets that are listed. The number of observations includes those that are marked for deletion but are not yet deleted.

Restriction: For certain SAS views, SAS cannot determine the number of observations. In these cases, SAS sets the value of the NOBS= variable to the largest positive integer value that is available in your operating environment.

Tip: At compilation time, SAS reads the descriptor portion of each data set and assigns the value of the NOBS= variable automatically. Thus, you can refer to the NOBS= variable before the SET statement. The variable is available in the DATA step but is not added to any output data set.

Interaction: The NOBS= and POINT= options are independent of each other.

Featured in: Example 10 on page 1016

OPEN=(IMMEDIATE | DEFER)
allows you to delay the opening of any concatenated SAS data sets until they are ready to be processed.

IMMEDIATE
during the compilation phase, opens all data sets that are listed in the SET statement.

Restriction: When you use the IMMEDIATE option KEY=, POINT=, and BY statement processing are mutually exclusive.

Tip: If a variable on a subsequent data set is of a different type (character versus numeric, for example) than that of the same-named variable on the first data set, the DATA step will stop processing and produce an error message.

DEFER
opens the first data set during the compilation phase, and opens subsequent data sets during the execution phase. When the DATA step reads and

processes all observations in a data set, it closes the data set and opens the next data set in the list.

Restriction: When you specify the DEFER option, you cannot use the KEY= statement option, the POINT= statement option, or the BY statement. These constructs imply either random processing or interleaving of observations from the data sets, which is not possible unless all data sets are open.

Requirement: You can use the DROP=, KEEP=, or RENAME= data set options to process a set of variables, but the set of variables that are processed for each data set must be identical. In most cases, if the set of variables defined by any subsequent data set differs from that defined by the first data set, SAS prints a warning message to the log but does not stop execution. Exceptions to this behavior are

1. If a variable on a subsequent data set is of a different type (character versus numeric, for example) than that of the same-named variable on the first data set, the DATA step will stop processing and produce an error message.
2. If a variable on a subsequent data set was not defined by the first data set in the SET statement, but was defined proviously in the DATA step program, the DATA step will stop processing and produce an error message. In this case, the value of the variable in previous iterations may be incorrect because the semantic behavior of SET requires this variable to be set to missing when processing the first observation of the first data set.

Default: IMMEDIATE

POINT=*variable*

specifies a temporary variable whose numeric value determines which observation is read. POINT= causes the SET statement to use random (direct) access to read a SAS data set.

Requirement: a STOP statement

Restriction: You cannot use POINT= with a BY statement, a WHERE statement, or a WHERE= data set option. In addition, you cannot use it with transport format data sets, data sets in sequential format on tape or disk, and SAS/ACCESS views or the SQL procedure views that read data from external files.

Restriction: You cannot use POINT= with KEY=.

Tip: You must supply the values of the POINT= variable. For example, you can use the POINT= variable as the index variable in some form of the DO statement.

Tip: The POINT= variable is available anywhere in the DATA step, but it is not added to any new SAS data set.

Featured in: Example 6 on page 1015 and Example 9 on page 1016

CAUTION:

Continuous loops can occur when you use the POINT= option. When you use the POINT= option, you must include a STOP statement to stop DATA step processing, programming logic that checks for an invalid value of the POINT= variable, or both. Because POINT= reads only those observations that are specified in the DO statement, SAS cannot read an end-of-file indicator as it would if the file were being read sequentially. Because reading an end-of-file indicator ends a DATA step automatically, failure to substitute another means of ending the DATA step when you use POINT= can cause the DATA step to go

into a continuous loop. If SAS reads an invalid value of the POINT= variable, it sets the automatic variable _ERROR_ to 1. Use this information to check for conditions that cause continuous DO-loop processing, or include a STOP statement at the end of the DATA step, or both. △

UNIQUE
causes a KEY= search always to begin at the top of the index for the data set that is being read.

Restriction: UNIQUE can only appear with the KEY= argument.

Explanation: By default, SET begins searching at the top of the index only when the KEY= value changes. If the KEY= value does not change on successive executions of the SET statement, the search begins by following the most recently retrieved observation. In other words, when consecutive duplicate KEY= values appear, the SET statement attempts a one-to-one match with duplicate indexed values in the data set that is being read. If more consecutive duplicate KEY= values are specified than exist in the data set that is being read, the extra duplicates are treated as not found.

Featured in: Example 8 on page 1015

See Also: For extensive examples, see "Examples" in *Combining and Modifying SAS Data Sets: Examples* .

Details

What SET Does Each time the SET statement is executed, SAS reads one observation into the program data vector. SET reads all variables and all observations from the input data sets unless you tell SAS to do otherwise. A SET statement can contain multiple data sets; a DATA step can contain multiple SET statements. See *Combining and Modifying SAS Data Sets: Examples* .

Uses The SET statement is flexible and has a variety of uses in SAS programming. These uses are determined by the options and statements that you use with the SET statement. They include

- reading observations and variables from existing SAS data sets for further processing in the DATA step
- concatenating and interleaving data sets, and performing one-to-one reading of data sets
- reading SAS data sets by using direct access methods.

BY Group Processing with SET Only one BY statement can accompany each SET statement in a DATA step. The BY statement should immediately follow the SET statement to which it applies. The data sets that are listed in the SET statement must be sorted by the values of the variables that are listed in the BY statement, or they must have an appropriate index. SET when it is used with a BY statement interleaves data sets. The observations in the new data set are arranged by the values of the BY variable or variables, and within each BY group, by the order of the data sets in which they occur. See Example 2 on page 1014 for an example of BY group processing with the SET statement.

Combining SAS Data Sets Use a single SET statement with multiple data sets that are specified to concatenate the specified data sets. That is, the number of observations in the new data set is the sum of the number of observations in the original data sets, and the order is all the observations from the first data set followed by all observations from the second data set, and so on. See Example 1 on page 1014 for an example of concatenating data sets.

Use a single SET statement with a BY statement to interleave the specified data sets. The observations in the new data set are arranged by the values of the BY variable or variables, and within each BY group, by the order of the data sets in which they occur. See Example 2 on page 1014 for an example of interleaving data sets.

Use multiple SET statements to perform one-to-one reading (also called one-to-one matching) of the specified data sets. The new data set contains all the variables from all the input data sets. The number of observations in the new data set is the number of observations in the smallest original data set. If the data sets contain common variables, the values that are read in from the last data set replace those read in from earlier ones. See Example 6 on page 1015, Example 7 on page 1015, and Example 8 on page 1015 for examples of one-to-one reading of data sets.

For extensive examples, see *Combining and Modifying SAS Data Sets: Examples* .

Comparisons

- SET reads an observation from an existing SAS data set. INPUT reads raw data from an external file or from in-stream data lines in order to create SAS variables and observations.
- Using the KEY= option with SET enables you to access observations nonsequentially in a SAS data set according to a value. Using the POINT= option with SET enables you to access observations nonsequentially in a SAS data set according to the observation number.

Examples

Example 1: Concatenating SAS Data Sets If more than one data set name appears in the SET statement, the resulting output data set is a concatenation of all the data sets that are listed. SAS reads all observations from the first data set, then all from the second data set, and so on until all observations from all the data sets have been read. This example concatenates the three SAS data sets into one output data set named FITNESS:

```
data fitness;
   set health exercise well;
run;
```

Example 2: Interleaving SAS Data Sets To interleave two or more SAS data sets, use a BY statement after the SET statement:

```
data april;
   set payable recvable;
   by account;
run;
```

Example 3: Reading a SAS Data Set In this DATA step, each observation in the data set NC.MEMBERS is read into the program data vector. Only those observations whose value of CITY is **Raleigh** are output to the new data set RALEIGH.MEMBERS:

```
data raleigh.members;
   set nc.members;
   if city='Raleigh';
run;
```

Example 4: Merging a Single Observation with All Observations in a SAS Data Set An observation to be merged into an exisitng data set can be one that is created by a SAS

procedure or another DATA step. In this example, the data set AVGSALES has only one observation:

```
data national;
      if _n_=1 then set avgsales;
      set totsales;
run;
```

Example 5: Reading from the Same Data Set More Than Once In this example, SAS treats each SET statement independently; that is, it reads from one data set as if it were reading from two separate data sets:

```
data drugxyz;
   set trial5(keep=sample);
   if sample>2;
   set trial5;
run;
```

For each iteration of the DATA step, the first SET statement reads one observation. The next time the first SET statement is executed, it reads the next observation. Each SET statement can read different observations with the same iteration of the DATA step.

Example 6: Combining One Observation with Many You can subset observations from one data set and combine them with observations from another data set by using direct access methods, as follows:

```
data south;
   set revenue;
   if region=4;
   set expense point=_n_;
run;
```

Example 7: Performing a Table-Lookup This example illustrates using the KEY= option to perform a table-lookup. The DATA step reads a primary data set that is named INVTORY and a lookup data set that is named PARTCODE. It uses the index PARTNO to read PARTCODE nonsequentially, by looking for a match between the PARTNO value in each data set. The purpose is to obtain the appropriate description, which is available only in the variable DESC in the lookup data set, for each part that is listed in the primary data set:

```
data combine;
   set invtory(keep=partno instock price);
   set partcode(keep=partno desc)  key=partno;
run;
```

Example 8: Performing a Table-Lookup When the Master File Contains Duplicate Observations This example uses the KEY= option to perform a table lookup. The DATA step reads a primary data set that is named INVTORY, which is indexed on PARTNO, and a lookup data set named PARTCODE. PARTCODE contains quantities of new stock (variable NEW_STK). The UNIQUE option ensures that, if there are any duplicate observations in INVTORY, values of NEW_STK are added only to the first observation of the group:

```
data combine;
   set partcode(keep=partno new_stk);
   set invtory(keep=partno instock price)
   key=partno/unique;
```

```
   instock=instock+new_stk;
run;
```

Example 9: Reading a Subset by Using Direct Access These statements select a subset of 50 observations from the data set DRUGTEST by using the POINT= option to access observations directly by number:

```
data sample;
   do obsnum=1 to 100 by 2;
      set drugtest point=obsnum;
      if _error_ then abort;
      output;
   end;
   stop;
run;
```

Example 10: Performing a Function Until the Last Observation Is Reached These statements use NOBS= to set the termination value for DO-loop processing. The value of the temporary variable LAST is the sum of the observations in SURVEY1 and SURVEY2:

```
do obsnum=1 to last by 100;
   set survey1 survey2 point=obsnum nobs=last;
   output;
end;
stop;
```

Example 11: Writing an Observation Only After All Observations Have Been Read This example uses the END= variable LAST to tell SAS to assign a value to the variable REVENUE and write an observation only after the last observation of RENTAL has been read:

```
set rental end=last;
totdays + days;
if last then
   do;
      revenue=totdays*65.78;
      output;
   end;
```

See Also

Statements:

"BY" on page 765

"DO" on page 789

"INPUT" on page 876

"MERGE" on page 930

"STOP" on page 1018

"UPDATE" on page 1023

"Rules for Words and Names" in *SAS Language Reference: Concepts*

"Reading, Modifying, and Combining SAS Data Sets" in *SAS Language Reference: Concepts*

Chapter 2, "Data Set Options," on page 5

SAS Macro Language: Reference

Combining and Modifying SAS Data Sets: Examples

SKIP

Creates a blank line in the SAS log

Valid: Anywhere

Category: Log Control

Syntax

SKIP <*n*>;

Without Arguments

Using SKIP without arguments causes SAS to create one blank line in the log.

Arguments

n

specifies the number of blank lines that you want to create in the log.

Tip: If the number specified is greater than the number of lines that remain on the page, SAS goes to the top of the next page.

Details

The SKIP statement itself does not appear in the log. You can use this statement in all methods of operation.

See Also

Statement:

"PAGE" on page 961

System Options:

"LINESIZE=" on page 1112
"PAGESIZE=" on page 1133

STOP

Stops execution of the current DATA step

Valid: in a DATA step
Category: Action
Type: Executable

Syntax

STOP;

Without Arguments

The STOP statement causes SAS to stop processing the current DATA step immediately and resume processing statements after the end of the current DATA step.

Details

SAS outputs a data set for the current DATA step. However, the observation being processed when STOP executes is not added. The STOP statement can be used alone or in an IF-THEN statement or SELECT group.

Use STOP with any features that read SAS data sets using random access methods, such as the POINT= option in the SET statement. Because SAS does not detect an end-of-file with this access method, you must include program statements to prevent continuous processing of the DATA step.

Comparisons

- When you use a windowing environment or other interactive methods of operation, the ABORT statement and the STOP statement both stop processing. The ABORT statement sets the value of the automatic variable _ERROR_ to 1, but the STOP statement does not.
- In batch or noninteractive mode, the two statements also have different effects. Use the STOP statement in batch or noninteractive mode to continue processing with the next DATA or PROC step.

Examples

Example 1: Basic Usage

```
stop;
```

```
□ if idcode=9999 then stop;

□ select (a);
     when (0) output;
     otherwise stop;
  end;
```

Example 2: Avoiding an Infinite Loop This example shows how to use STOP to avoid an infinite loop within a DATA step when you are using random access methods:

```
data sample;
   do sampleobs=1 to totalobs by 10;
      set master.research point=sampleobs
                          nobs=totalobs;
      output;
   end;
   stop;
run;
```

See Also

Statements:

"ABORT" on page 752

POINT= option in the SET statement on page 1012

Sum

Adds the result of an expression to an accumulator variable

Valid: in a DATA step

Category: Action

Type: Executable

Syntax

variable+expression;

Arguments

variable

specifies the name of the accumulator variable, which contains a numeric value.

Tip: The variable is automatically set to 0 before SAS reads the first observation. The variable's value is retained from one iteration to the next, as if it had appeared in a RETAIN statement.

Tip: To initialize a sum variable to a value other than 0, include it in a RETAIN statement with an initial value.

expression

is any SAS expression.

Tip: The expression is evaluated and the result added to the accumulator variable.

Tip: SAS treats an expression that produces a missing value as zero.

Comparisons

The sum statement is equivalent to using the SUM function and the RETAIN statement, as shown here:

```
retain variable 0;
variable=sum(variable,expression);
```

Examples

Here are examples of sum statements that illustrate various expressions:

- `balance+(-debit);`
- `sumxsq+x*x;`
- `nx+(x ne .);`
- `if status='ready' then OK+1;`

See Also

Function:
"SUM" on page 556
Statement:
"RETAIN" on page 999

TITLE

Specifies title lines for SAS output

Valid: anywhere
Category: Output Control

Syntax

TITLE <*n*> <'*text*' | "*text*">;

Without Arguments

Using TITLE without arguments cancels all existing titles.

Arguments

n
specifies the relative line that contains the title line.

Range: 1 - 10

Tip: The title line with the highest number appears on the bottom line. If you omit *n*, SAS assumes a value of 1. Therefore, you can specify TITLE or TITLE1 for the first title line.

Tip: You can create titles that contain blank lines between the lines of text. For example, if you specify text with a TITLE statement and a TITLE3 statement, there will be a blank line between the two lines of text.

'*text*' | "*text*"
specifies text that is enclosed in single or double quotation marks.

Tip: For compatibility with previous releases, SAS accepts some text without quotation marks.

Tip: When writing new programs or updating existing programs, *always* surround text with quotation marks.

See Also: For more information about including quotation marks as part of the title, see the "Expressions" chapter in *SAS Language Reference: Concepts*.

Details

In a DATA Step or PROC Step A TITLE statement takes effect when the step or RUN group with which it is associated executes. Once you specify a title for a line, it is used for all subsequent output until you cancel the title or define another title for that line. A TITLE statement for a given line cancels the previous TITLE statement for that line and for all lines with larger *n* numbers.

Operating Environment Information: The maximum title length that is allowed depends on your operating environment and the value of the LINESIZE= system option. Refer to the SAS documentation for your operating environment for more information. △

Customizing Titles in a PROC step You can customize titles by inserting BY variable values (#BYVAL*n*), BY variable names (#BYVAR*n*), or BY lines (#BYLINE) in titles that are specified in PROC steps. Embed the items in the specified title text string at the position where you want the substitution text to appear.

#BYVAL*n* | #BYVAL(*variable-name*)
substitutes the current value of the specified BY variable for #BYVAL in the text string and displays the value in the title. Specify the variable with one of the following:

n
specifies which variable in the BY statement that #BYVAL should use. The value of *n* indicates the position of the variable in the BY statement.

Example: #BYVAL2 specifies the second variable in the BY statement.

variable-name
names the BY variable.

Example: #BYVAL(YEAR) specifies the BY variable, YEAR.

Tip: *Variable-name* is not case sensitive.

#BYVAR*n* | #BYVAR(*variable-name*)
substitutes the name of the BY variable or label that is associated with the variable (whatever the BY line would normally display) for #BYVAR in the text string and displays the name or label in the title. Specify the variable with one of the following:

n
specifies which variable in the BY statement that #BYVAR should use. The value of *n* indicates the position of the variable in the BY statement.

Example: #BYVAR2 specifies the second variable in the BY statement.

variable-name
names the BY variable.

Example: #BYVAR(SITES) specifies the BY variable SITES.

Tip: *Variable-name* is not case sensitive.

#BYLINE
substitutes the entire BY line without leading or trailing blanks for #BYLINE in the text string and displays the BY line in the title.

Tip: #BYLINE produces output that contains a BY line at the top of the page unless you suppress it by using NOBYLINE in an OPTIONS statement.

See Also: For more information on NOBYLINE, see "BYLINE" on page 1066.

In a PROC step Follow these rules when you use #BYVAR, #BYVAL, and #BYLINE in the TITLE statement of a PROC step:

- Specify the variable that is used by #BYVAR or #BYVAL in the BY statement.
- Insert #BYVAL and #BYVAR in the specified title text string at the position where you want the substitution text to appear.
- Follow #BYVAL and #BYVAR with a delimiting character, either a space or other nonalphanumeric character (for example, a quotation mark) that ends the text string.
- If you want the #BYVAR or #BYVAL substitution to be followed immediately by other text, with no delimiter, use a trailing dot (as with macro variables).

Comparisons

You can also create titles with the TITLES window.

Examples

- This statement suppresses a title on line *n* and all lines after it:

```
titlen;
```

- These are examples of TITLE statements:
  - `title 'First Draft';`
  - `title2 "Year's End Report";`
  - `title2 'Year''s End Report';`
- These statements illustrate #BYVAL, #BYVAR, and #BYLINE.
  - `title 'Quarterly Sales for #byval(site)';`
  - `title 'Annual Costs for #byvar2';`
  - `title 'Data Group #byline';`

See Also

Statement:

"FOOTNOTE" on page 841

System Option:

"LINESIZE=" on page 1112

UPDATE

Updates a master file by applying transactions

Valid: in a DATA step
Category: File-handling
Type: Executable

Syntax

UPDATE *master-data-set*<(*data-set-options*)> *transaction-data-set*<(*data-set-options*)>
<END=*variable*>
<UPDATEMODE=
MISSINGCHECK|NOMISSINGCHECK>;
BY *by-variable*;

Arguments

master-data-set
names the SAS data set used as the master file.

Range: The name can be a one-level name (for example, FITNESS), a two-level name (for example, IN.FITNESS), or one of the special SAS data set names.

See Also: SAS Names and Words in *SAS Language Reference: Concepts*.

(data-set-options)
specifies actions SAS is to take when it reads variables into the DATA step for processing.

Requirements: *Data-set-options* must appear within parentheses and follow a SAS data set name.

Tip: Dropping, keeping, and renaming variables is often useful when you update a data set. Renaming like-named variables prevents the second value that is read from over-writing the first one. By renaming one variable, you make the values of both of them available for processing, such as comparing.

Featured in: Example 2 on page 1025

See Also: A list of data set options to use with input data sets in "Data Set Options by Category" on page 7 .

transaction-data-set
names the SAS data set that contains the changes to be applied to the master data set.

Range: The name can be a one-level name (for example, HEALTH), a two-level name (for example, IN.HEALTH), or one of the special SAS data set names.

END=*variable*
creates and names a temporary variable that contains an end-of-file indicator. This variable is initialized to 0 and is set to 1 when UPDATE processes the last observation. This variable is not added to any data set.

UPDATEMODE=MISSINGCHECK
UPDATEMODE=NOMISSINGCHECK
specifies if missing variable values in a transaction data set are to be allowed to replace existing variable values in a master data set.

MISSINGCHECK
performs a check that prevents missing variable values in a transaction data set from replacing values in a master data set.

NOMISSINGCHECK
prevents a check and, therefore, allows missing variable values in a transaction data set to replace values in a master data set.

Tip: Special missing values, however, are the exception and will replace values in the master data set even when MISSINGCHECK (the default) is in effect.

Default: MISSINGCHECK

Details

Requirements

- The UPDATE statement must be accompanied by a BY statement that specifies the variables by which observations are matched.
- The BY statement should immediately follow the UPDATE statement to which it applies.
- The data sets listed in the UPDATE statement must be sorted by the values of the variables listed in the BY statement, or they must have an appropriate index.
- Each observation in the master data set should have a unique value of the BY variable or BY variables. If there are multiple values for the BY variable, only the first observation with that value is updated. The transaction data set can contain more than one observation with the same BY value. (Multiple transaction observations are all applied to the master observation before it is written to the output file.)

Transaction Data Sets Usually, the master data set and the transaction data set contain the same variables. However, to reduce processing time, you can create a transaction data set that contains only those variables that are being updated. The transaction data set can also contain new variables to be added to the output data set.

The output data set contains one observation for each observation in the master data set. If any transaction observations do not match master observations, they become new observations in the output data set. Observations that are not to be updated can be omitted from the transaction data set. See “Reading, Modifying, and Combining SAS Data Sets” in *SAS Language Reference: Concepts*.

Missing Values By default the UPDATEMODE=MISSINGCHECK option is in effect, so missing values in the transaction data set do *not* replace existing values in the master data set. Therefore, if you want to update some but not all variables and if the variables you want to update differ from one observation to the next, set to missing those variables that are not changing. If you want missing values in the transaction

data set to replace existing values in the master data set, use UPDATEMODE=NOMISSINGCHECK.

Even when UPDATEMODE=MISSINGCHECK is in effect, you can replace existing values with missing values by using special missing value characters in the transaction data set. To create the transaction data set, use the MISSING statement in the DATA step. If you define one of the special missing values **A** through **Z** for the transaction data set, SAS updates numeric variables in the master data set to that value.

If you want the resulting value in the master data set to be a regular missing value, use a single underscore (_) to represent missing values in the transaction data set. The resulting value in the master data set will be a period (.) for missing numeric values and a blank for missing character values.

For more information about defining and using special missing value characters, see "MISSING" on page 932 .

Comparisons

- Both UPDATE and MERGE can update observations in a SAS data set.
- MERGE automatically replaces existing values in the first data set with missing values in the second data set. UPDATE, however, does not do so by default. To cause UPDATE to overwrite existing values in the master data set with missing ones in the transaction data set, you must use UPDATEMODE=NOMISSINGCHECK.
- UPDATE changes or updates the values of selected observations in a master file by applying transactions. UPDATE can also add new observations.

Examples

Example 1: Basic Updating These program statements create a new data set (OHIO.QTR1) by applying transactions to a master data set(OHIO.JAN). The BY variable STORE must appear in both OHIO.JAN and OHIO.WEEK4, and its values in the master data set should be unique:

```
data ohio.qtr1;
   update ohio.jan ohio.week4;
   by store;
run;
```

Example 2: Updating By Renaming Variables This example shows renaming a variable in the FITNESS data set so that it will not overwrite the value of the same variable in the program data vector. Also, the WEIGHT variable is renamed in each data set and a new WEIGHT variable is calculated. The master data set and the transaction data set are listed before the code that performs the update:

```
Master Data Set
            HEALTH

OBS    ID     NAME     TEAM    WEIGHT

 1    1114    sally    blue      125
 2    1441    sue      green     145
 3    1750    joey     red       189
 4    1994    mark     yellow    165
 5    2304    joe      red       170
```

```
Transaction Data Set
            FITNESS

OBS   ID     NAME     TEAM    WEIGHT

 1   1114    sally    blue      119
 2   1994    mark     yellow    174
 3   2304    joe      red       170
```

```
options nodate pageno=1 linesize=80 pagesize=60;

   /* Sort both data sets by ID */
proc sort data=health;
   by id;
run;
proc sort data=fitness;
   by id;
run;

   /* Update Master with Transaction */
data health2;
   length STATUS $11;
   update health(rename=(weight=ORIG) in=a)
          fitness(drop=name team in=b);
   by id ;
   if a and b then
      do;
         CHANGE=abs(orig - weight);
         if weight<orig then status='loss';
         else if weight>orig then status='gain';
         else status='same';
      end;
   else status='no weigh in';
run;

options nodate ls=78;

proc print data=health2;
   title 'Weekly Weigh-in Report';
run;
```

Output 6.11

```
                    Weekly Weigh-in Report              1

 OBS STATUS        ID   NAME   TEAM    ORIG  WEIGHT  CHANGE

  1  loss         1114  sally  blue     125    119       6
  2  no weigh in  1441  sue    green    145      .       .
  3  no weigh in  1750  joey   red      189      .       .
  4  gain         1994  mark   yellow   165    174       9
  5  same         2304  joe    red      170    170       0
```

Example 3: Updating with Missing Values This example illustrates the DATA steps used to create a master data set PAYROLL and a transaction data set INCREASE that contains regular and special missing values:

```
options nodate pageno=1 linesize=80 pagesize=60;

   /* Create the Master Data Set */
data payroll;
   input ID SALARY;
   datalines;
011 245
026 269
028 374
034 333
057 582
;

   /* Create the Transaction Data Set */
data increase;
   input ID SALARY;
   missing A _;
   datalines;
011 376
026 .
028 374
034 A
057 _
;

   /* Update Master with Transaction */
data newpay;
   update payroll increase;
   by id;
run;
proc print data=newpay;
   title 'Updating with Missing Values';
run;
```

Output 6.12

```
      Updating with Missing Values              1

          OBS     ID     SALARY

            1     1011      376
            2     1026      269    <=== value remains 269
            3     1028      374
            4     1034        A    <=== special missing value
            5     1057        .    <=== regular missing value
```

See Also

Statements:

"BY" on page 765
"MERGE" on page 930
"MISSING" on page 932
"MODIFY" on page 933
"SET" on page 1010

System Option:

"MISSING=" on page 1117

"Reading, Modifying, and Combining SAS Data Sets" in *SAS Language Reference: Concepts*

Chapter 2, "Data Set Options," on page 5

WHERE

Selects observations from SAS data sets that meet a particular condition

Valid: in DATA and PROC steps
Category: Action
Type: Declarative

Syntax

WHERE *where-expression-1*
< *logical-operator where-expression-n*>;

Arguments

where-expression
is an arithmetic or logical expression that generally consists of a sequence of operands and operators.

Tip: The operands and operators described in the next several sections are also valid for the WHERE= data set option.

Tip: You can specify multiple where-expressions.

logical-operator
can be AND, AND NOT, OR, or OR NOT.

Details

General Information Using the WHERE statement may improve the efficiency of your SAS programs because SAS is not required to read all observations from the input data set.

The WHERE statement cannot be executed conditionally; that is, you cannot use it as part of an IF-THEN statement.

WHERE statements can contain multiple WHERE expressions that are joined by logical operators.

Note: Using indexed SAS data sets can significantly improve performance when you use WHERE expressions to access a subset of the observations in a SAS data set. For more information about indexes, see "Indexes" in the "SAS Data Files" chapter of *SAS Language Reference: Concepts* for a complete discussion of WHERE-expression processing with indexed data sets and a list of guidelines to consider before you index your SAS data sets. △

In DATA Steps The WHERE statement applies to all data sets in the preceding SET, MERGE, MODIFY, or UPDATE statement, and variables that are used in the WHERE statement must appear in all of those data sets. You cannot use the WHERE statement with programming statements that select observations by observation number, such as the OBS= data set option and the POINT= option in the SET and MODIFY statements. When you use the WHERE statement, the FIRSTOBS= data set option must be 1. You cannot use the WHERE statement to select records from an external file that contains raw data, nor can you use the WHERE statement within the same DATA step in which you read in-stream data with a CARDS or DATALINES statement.

For each iteration of the DATA step, the first operation the SAS System performs in each execution of a SET, MERGE, MODIFY, or UPDATE statement is to determine whether the observation in the input data set meets the condition of the WHERE statement. The WHERE statement takes effect immediately after the input data set options are applied and before any other statement in the DATA step is executed. If a DATA step combines observations using a WHERE statement with a MERGE, MODIFY, or UPDATE statement, SAS selects observations from each input data set before it combines them.

WHERE and BY in a DATA Step If a DATA step contains both a WHERE statement and a BY statement, the WHERE statement executes *before* BY groups are created. Therefore, BY groups reflect groups of observations in the subset of observations that are selected by the WHERE statement, not the actual BY groups of observations in the original input data set.

For a complete discussion of BY-group processing, see "BY-Group Processing" in *SAS Language Reference: Concepts*.

In PROC Steps You can use the WHERE statement with any SAS procedure that reads a SAS data set. The WHERE statement is useful for subsetting the original data set for processing by the procedure. The *SAS Procedures Guide* documents the action of the WHERE statement only in those procedures for which you can specify more than one data set. In all other cases, the WHERE statement performs as documented here.

Use of Indexes A DATA or PROC step attempts to use an available index to optimize the selection of data when an indexed variable is used in combination with one of the following:

- the BETWEEN-AND operator
- the comparison operators, with or without the colon modifier
- the CONTAINS operator
- the IS NULL and IS NOT NULL operators
- the LIKE operator
- the TRIM function
- the SUBSTR function, in some cases.

SUBSTR requires the following arguments:

```
where substr(variable,position,length)
 ='character-string';
```

An index is used in processing when the arguments of the SUBSTR function meet all of the following conditions:

- *position* is equal to 1
- *length* is less than or equal to the length of *variable*
- *length* is equal to the length of *character-string*.

Operands Used in WHERE Expressions Operands include

- constants
- time and date values
- values of variables that are obtained from the SAS data sets
- values created within the WHERE expression itself.

You cannot use variables that are created within the DATA step (for example, FIRST.*variable*, LAST.*variable*, _N_, or variables that are created in assignment statements) in a WHERE expression because the WHERE statement is executed before the SAS System brings observations into the DATA or PROC step. When WHERE expressions contain comparisons, the unformatted values of variables are compared.

Use operands in WHERE statements as in the following examples:

- where score>50;
- where date>='01jan1999'd and time>='9:00't;
- where state='Mississippi';

As in other SAS expressions, the names of numeric variables can stand alone. SAS treats values of 0 or missing as false; other values are true. These examples are WHERE expressions that contain the numeric variables EMPNUM and SSN:

- where empnum;
- where empnum and ssn;

Character literals or the names of character variables can also stand alone in WHERE expressions. If you use the name of a character variable by itself as a WHERE expression, SAS selects observations where the value of the character variable is not blank.

Operators Used in the WHERE Expression You can include both SAS operators and special WHERE expression operators in the WHERE statement. For a complete list of the operators, see Table 6.8 on page 1030. For the rules SAS follows when it evaluates WHERE expressions, see "WHERE Processing" in *SAS Language Reference: Concepts*.

Table 6.8 WHERE Statement Operators

Operator Type	Symbol or Mnemonic	Description
Arithmetic		
	*	multiplication
	/	division
	+	addition
	–	subtraction

Operator Type	Symbol or Mnemonic	Description
	**	exponentiation
Comparison		
	= or EQ	equal to
	^, =, <>, or NE [1]	not equal to
	> or GT	greater than
	< or LT	less than
	>= or GE	greater than or equal to
	<= or LE	less than or equal to
	IN	equal to one of a list
Logical (Boolean)		
	& or AND	logical and
	| or OR[2]	logical or
	~,^ , ¬, or NOT[3]	logical not
Other		
	||[4]	concatenation of character variables
	( )	indicate order of evaluation
	+ prefix	positive number
	– prefix	negative number
WHERE Expression Only		
	BETWEEN–AND	an inclusive range
	? or CONTAINS	a character string
	IS NULL or IS MISSING	missing values
	LIKE	match patterns
	=*	sounds-like
	SAME-AND	add clauses to an existing WHERE statement without retyping original one

1 The caret (^), tilde (~), and the not sign (¬)all indicate a logical not. Use the character available on your keyboard, or use the mnemonic equivalent.
2 The OR symbol (|), broken vertical bar (|), and exclamation point (!) all indicate a logical or. Use the character available on your keyboard, or use the mnemonic equivalent.
3 The caret (^), tilde (~), and the not sign (¬)all indicate a logical not. Use the character available on your keyboard, or use the mnemonic equivalent.
4 Two OR symbols (| |), two broken vertical bars (| |), or two exclamation points (!!) indicate concatenation. Use the character available on your keyboard.

You can use the colon modifier (:) with any of the comparison operators. See Appendix 2, “SAS Operators,” on page 1189 for more information about the colon modifier.

Comparisons

- You can use the WHERE command in SAS/FSP software to subset data for editing and browsing. You can use both the WHERE statement and WHERE= data set option in windowing procedures and in conjunction with the WHERE command.

- To select observations from individual data sets when a SET, MERGE, MODIFY, or UPDATE statement specifies more than one data set, apply a WHERE= data set option to each data set. In the DATA step, if a WHERE statement and a WHERE= data set option apply to the same data set, SAS uses the data set option and ignores the statement.
- The most important differences between the WHERE statement in the DATA step and the subsetting IF statement are as follows:
  - The WHERE statement selects observations *before* they are brought into the program data vector, making it a more efficient programming technique. The subsetting IF statement works on observations after they are read into the program data vector.
  - The WHERE statement can produce a different data set from the subsetting IF when a BY statement accompanies a SET, MERGE, or UPDATE statement. The different data set occurs because SAS creates BY groups before the subsetting IF statement selects but after the WHERE statement selects.
  - The WHERE statement cannot be executed conditionally as part of an IF statement, but the subsetting IF statement can.
  - The WHERE statement selects observations in SAS data sets only, whereas the subsetting IF statement selects observations from an existing SAS data set or from observations that are created with an INPUT statement.
  - The subsetting IF statement cannot be used in SAS windowing procedures to subset observations for browsing or editing.
- Do not confuse the WHERE statement with the DROP or KEEP statement. The DROP and KEEP statements select variables for processing. The WHERE statement selects observations.

Examples

Example 1: Basic WHERE Statement Usage This DATA step produces a SAS data set that contains only observations from data set CUSTOMER in which the value for NAME begins with **Mac** and the value for CITY is **Charleston** or **Atlanta**.

```
data testmacs;
   set customer;
   where substr(name,1,3)='Mac' and
         (city='Charleston' or city='Atlanta');
run;
```

Example 2: Using Operators Available Only in the WHERE Statement

- Using BETWEEN-AND:

```
where empnum between 500 and 1000;
```

- Using CONTAINS:

```
where company ? 'bay';
where company contains 'bay';
```

- Using IS NULL and IS MISSING:

```
where name is null;
where name is missing;
```

- Using LIKE to select all names that start with the letter D:

```
where name like 'D%';
```

□ Using LIKE to match patterns from a list of the following names:

Diana
Diane
Dianna
Dianthus
Dyan

WHERE Statement	Name Selected
where name like 'D_an';	Dyan
where name like 'D_an_';	Diana, Diane
where name like 'D_an__';	Dianna
where name like 'D_an%';	all names from list

□ Using the Sounds-like Operator to select names that sound like "Smith":

```
where lastname=*'Smith';
```

□ Using SAME-AND:

```
where year>1991;
...more SAS statements...
where same and year<1999;
```

In this example, the second WHERE statement is equivalent to the following WHERE statement

```
where year>1991 and year<1999;
```

:

See Also

Data Set Option:
"WHERE=" on page 43
Statement:
"IF, Subsetting" on page 847
SAS Guide to the SQL Procedure: Usage and Reference
SAS/IML Software: Usage and Reference
SAS Procedures Guide
"SAS Indexes" in *SAS Language Reference: Concepts*
"WHERE Processing" in *SAS Language Reference: Concepts*
"BY-Group Processing" in *SAS Language Reference: Concepts*
Beatrous, S. & Clifford, W. (1998), "Sometimes You Do Get What You Want: SAS I/O Enhancements in Version 7," *Proceedings of the Twenty–third Annual SAS Users Group International Conference*, 23.

WINDOW

Creates customized windows for your applications

Valid: in a DATA step
Category: Window Display
Type: Declarative

Syntax

WINDOW *window* <*window-options*> *field-definition(s)*;

WINDOW *window* <*window-options*> *group-definition(s)*;

Arguments

window
names the window.

Restriction: Window names must conform to SAS naming conventions.

window-options
specifies characteristics of the window as a whole. Specify all *window-options* before any field or GROUP= specifications. *Window-options* can include

COLOR=*color*
specifies the color of the window background for operating environments that have this capability. In other operating environments, this option affects the color of the window border. The following colors are available:

BLACK

BLUE

BROWN

CYAN

GRAY

GREEN

MAGENTA

ORANGE

PINK

RED

WHITE

YELLOW

Default: If you do not specify a color with the COLOR= option, the window's background color is device-dependent instead of black, and the color of a field is device-dependent instead of white.

Tip: The representation of colors may vary, depending on the monitor being used. COLOR= has no effect on monochrome monitors.

COLUMNS=*columns*
specifies the number of columns in the window.

Default: The window fills all remaining columns in the display; the number of columns that are available depends on the type of monitor that is being used.

ICOLUMN=*column*
specifies the initial column within the display at which the window is displayed.

Default: SAS displays the window at column 1.

IROW=*row*
specifies the initial row (or line) within the display at which the window is displayed.

Default: SAS displays the window at row 1.

KEYS=<<*libref.*>*catalog.*>*keys-entry*
specifies the name of a KEYS entry that contains the function key definitions for the window.

Default: SAS uses the current function key settings that are defined in the KEYS window.

Tip: If you specify only an entry name, SAS looks in the SASUSER.PROFILE catalog for a KEYS entry of the name that is specified. You can also specify the three-level name of a KEYS entry, in the form

```
libref.catalog.keys-entry
```

Tip: To create a set of function key definitions for a window, use the KEYS window. Define the keys as you want, and use the SAVE command to save the definitions in the SASUSER.PROFILE catalog or in a SAS data library and catalog that you specify.

MENU=<<*libref.*>*catalog.*>*pmenu-entry*
specifies the name of a pull-down menu (pmenu) you have built with the PMENU procedure.

Tip: If you specify only an entry name, SAS looks in the SASUSER.PROFILE catalog for a PMENU entry of the name specified. You can also specify the three-level name of a PMENU entry in the form

```
libref.catalog.pmenu-entry
```

ROWS=*rows*
specifies the number of rows (or lines) in the window.

Default: The window fills all remaining rows in the display.

Tip: The number of rows that are available depends on the type of monitor that is being used.

field-definition
identifies and describes a variable or character string to be displayed in a window or within a group of related fields.

Tip: A window or group can contain any number of fields, and you can define the same field in several groups or windows.

Tip: You can specify multiple *field-definitions*.

See Also: The form of *field-definition* is given in “Field Definitions” on page 1036.

group-definition
names a group and defines all fields within a group. A group definition consists of two parts: the GROUP= option and one or more field definitions.

GROUP=*group*
names a group of related fields.

Restriction: *group* must be a SAS name.

Default: A window contains one unnamed group of fields.

Tip: When you refer to a group in a DISPLAY statement, write the name as *window.group*.

Tip: A group contains all fields in a window that you want to display at the same time. Display various groups of fields within the same window at different times by naming each group. Choose the group to appear by specifying *window.group* in the DISPLAY statement.

Tip: Specifying several groups within a window prevents repetition of window options that do not change and helps you to keep track of related displays. For example, if you are defining a window to check data values, arrange the display of variables and messages for most data values in the data set in a group that is named STANDARD. Arrange the display of different messages in a group that is named CHECKIT that appears when data values meet the conditions that you want to check.

Details

Operating Environment Information: The WINDOW statement has some functionality that is specific to your operating environment. For details, see the SAS documentation for your operating environment. △

You can use the WINDOW statement in the SAS windowing environment, in interactive line mode, or in noninteractive mode to create customized windows for your applications.* Windows that you create can display text and accept input; they have command and message lines. The window name appears at the top of the window. Use commands and function keys with windows that you create. A window definition remains in effect only for the DATA step that contains the WINDOW statement.

Define a window before you display it. Use the DISPLAY statement to display windows that are created with the WINDOW statement. For information about the DISPLAY statement, see "DISPLAY" on page 785.

Field Definitions Use a field definition to identify a variable or a character string to be displayed, its position, and its attributes. Enclose character strings in quotation marks. The position of an item is its beginning row (or line) and column. Attributes include color, whether you can enter a value into the field, and characteristics such as highlighting.

You can define a field to contain a variable value or a character string, but not both. The form of a field definition for a variable value is

<row column> variable <format> options

The form for a character string is

<row column> 'character-string' options

The elements of a field definition are described here.

row column
identifies the position of the variable or character string.

Default: If you omit *row* in the first field of a window or group, SAS uses the first row of the window; if you omit *row* in a later field specification, SAS continues on the row that contains the previous field. If you omit *column*, SAS uses column 1 (the left border of the window).

* You cannot use the WINDOW statement in batch mode because no terminal is connected to a batch executing process.

Tip: Although you can specify either *row* or *column* first, the examples in this book show the row first.

SAS keeps track of its position in the window with a pointer. For example, when you tell SAS to write a variable's value in the third column of the second row of a window, the pointer moves to row 2, column 3 to write the value. Use the pointer controls that are listed here to move the pointer to the appropriate position for a field.

In a field definition, *row* can be one of these row pointer controls:

#n
specifies row *n* within the window.
Range: *n* must be a positive integer.

#numeric-variable
specifies the row within the window that is given by the value of *numeric-variable*.
Restriction: *#numeric-variable* must be a positive integer. If the value is not an integer, the decimal portion is truncated and only the integer is used.

#(*expression*)
specifies the row within the window that is given by the value of *expression*.
Restriction: *expression* can contain array references and must evaluate to a positive integer.
Restriction: Enclose *expression* in parentheses.

/
moves the pointer to column 1 of the next row.

In a field definition, *column* can be one of these column pointer controls:

@n
specifies column *n* within the window.
Restriction: *n* must be a positive integer.

@numeric-variable
specifies the column within the window that is given by the value of *numeric-variable*.
Restriction: *numeric-variable* must be a positive integer. If the value is not an integer, the decimal portion is truncated and only the integer is used.

@(*expression*)
specifies the column within the window that is given by the value of *expression*.
Restriction: *expression* can contain array references and must evaluate to a positive integer.
Restriction: Enclose *expression* in parentheses.

+n
moves the pointer *n* columns.
Range: *n* must be a positive integer.

+numeric-variable
moves the pointer the number of columns that is given by the *numeric-variable*.
Restriction: *+numeric-variable* must be a positive or negative integer. If the value is not an integer, the decimal portion is truncated and only the integer is used.

variable
names a variable to be displayed or to be assigned the value that you enter at that position when the window is displayed.

Tip: *variable* can be the name of a variable or of an array reference.

Tip: To allow a variable value in a field to be displayed but not changed by the user, use the PROTECT= option (described later in this section). You can also protect an entire window or group for the current execution of the DISPLAY statement by specifying the NOINPUT option in the DISPLAY statement.

Tip: If a field definition contains the name of a new variable, that variable is added to the data set that is being created (unless you use a KEEP or DROP specification).

format
gives the format for the variable.

Default: If you omit *format*, SAS uses an informat and format that are specified elsewhere (for example, in an ATTRIB, INFORMAT, or FORMAT statement or permanently stored with the data set) or a SAS default informat and format.

Tip: If a field displays a variable that cannot be changed (that is, you use the PROTECT=YES option), *format* can be any SAS format or a format that you define with the FORMAT procedure.

Tip: If a field can both display a variable and accept input, you must either specify the informat in an INFORMAT or ATTRIB statement or use a SAS format such as $CHAR. or TIME. that has a corresponding informat.

Tip: If a format is specified, the corresponding informat is assigned automatically to fields that can accept input.

Tip: A format and an informat in a WINDOW statement override an informat and a format that are specified elsewhere.

'*character-string*'
contains the text of a character string to be displayed.

Restriction: The character string must be enclosed in quotation marks.

Restriction: You cannot enter a value in a field that contains a character string.

options
include any of the following:

ATTR=*highlighting-attribute*
controls these highlighting attributes of the field:

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.

Alias: A=

Tip: To specify more than one highlighting attribute, use the form

ATTR=(*highlighting-attribute-1*, . . .)

Tip: The highlighting attributes that are available depend on the type of monitor that you use.

AUTOSKIP=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.

YES — specifies that the cursor moves automatically to the next unprotected field.

NO — specifies that the cursor does not move automatically.

Alias: AUTO=

Default: NO

COLOR=*color*
specifies a color for the variable or character string. The following colors are available:

BLACK

BLUE

BROWN

CYAN

GRAY

GREEN

MAGENTA

ORANGE

PINK

RED

WHITE

YELLOW

Alias: C=

Default: WHITE

Tip: The representation of colors may vary, depending on the monitor you use.

Tip: COLOR= has no effect on monochrome monitors.

DISPLAY=YES | NO
controls whether the contents of a field are displayed.

YES — specifies that SAS displays characters in a field as you type them in.

NO — specifies that the entered characters are not displayed.

Default: YES

PERSIST=YES | NO
controls whether a field is displayed by all executions of a DISPLAY statement in the same iteration of the DATA step until the DISPLAY statement contains the BLANK option.

YES — specifies that each execution of the DISPLAY statement displays all previously displayed contents of the field as well as those that are scheduled for display by the current DISPLAY statement. If the new contents overlap persisting contents, the persisting contents are no longer displayed.

NO specifies that each execution of a DISPLAY statement displays only the current contents of the field.

Default: NO

Tip: PERSIST= is most useful when the position of a field changes in each execution of a DISPLAY statement.

Featured in: Example 3 on page 1042

PROTECT=YES | NO
controls whether information can be entered into a field.

YES specifies that you cannot enter information.

NO specifies that you can enter information.

Alias: P=

Default: No

Tip: Use PROTECT= only for fields that contain variables; fields that contain text are automatically protected.

REQUIRED=YES | NO
controls whether a field can be left blank.

NO specifies that you can leave the field blank.

YES specifies that you must enter a value in the field.

Default: NO

Tip: If you try to leave a field blank that was defined with REQUIRED=YES, SAS does not allow you to input values in any subsequent fields in the window.

Automatic Variables The WINDOW statement creates two automatic SAS variables: _CMD_ and _MSG_.

CMD contains the last command from the window's command line that was not recognized by the window.

Tip: _CMD_ is a character variable of length 80; its value is set to ''(blank) before each execution of a DISPLAY statement.

Featured in: Example 4 on page 1043

MSG contains a message that you specify to be displayed in the message area of the window.

Tip: _MSG_ is a character variable with length 80; its value is set to ''(blank) after each execution of a DISPLAY statement.

Featured in: Example 4 on page 1043

Displaying Windows The DISPLAY statement enables you to display windows. Once you display a window, the window remains visible until you display another window over it or until the end of the DATA step. When you display a window that contains fields into which you can enter values, either enter a value or press ENTER at *each* unprotected field to cause SAS to proceed to the next display. While a window is being displayed, you can use commands and function keys to view other windows, change the size of the current window, and so on. SAS execution proceeds to the next display only after you have pressed ENTER in all unprotected fields.

A DATA step that contains a DISPLAY statement continues execution until

- the last observation that is read by a SET, MERGE, MODIFY, UPDATE, or INPUT statement has been processed
- a STOP or ABORT statement is executed

- an END command executes.

Comparisons

- The WINDOW statement creates a window, and the DISPLAY statement displays it.
- The %WINDOW and %DISPLAY statements in the macro language create and display windows that are controlled by the macro facility.

Examples

Example 1: Creating a Single Window This DATA step creates a window with a single group of fields:

```
data _null_;
   window start
          #9  @26 'WELCOME TO THE SAS SYSTEM'
                  color=black
          #12  @19 'THIS PROGRAM CREATES'
          #12  @40 'TWO SAS DATA SETS'
          #14 @26 'AND USES THREE PROCEDURES'
          #18 @27 'Press ENTER to continue';
    display start;
    stop;
run;
```

The START window fills the entire display. The first line of text is black. The other three lines are the default for your operating environment. The text begins in the column that you specified in your program. The START window does not require you to input any values. However, to exit the window do one of the following:

- Press ENTER to cause DATA step execution to proceed to the STOP statement.
- Issue the END command.

If you omit the STOP statement from this program, the DATA step executes endlessly until you execute END from the window, either with a function key or from the command line. (Because this DATA step does not read any observations, SAS cannot detect an end-of-file to end DATA step execution.)

Example 2: Displaying Two Windows Simultaneously The following statements assign news articles to reporters. The list of article topics is stored as variable art in SAS data set category.article. This application allows you to assign each topic to a writer and to view the accumulating assignments. The program creates a new SAS data set named Assignment.

```
libname category 'SAS-data-library';

data Assignment;
   set category.article end=final;
   drop a b j s o;
   window Assignment irow=1 rows=12 color=white
          #3 @10 'Article:' +1 art protect=yes
          'Name:' +1 name $14.;
   window Showtotal irow=20 rows=12 color=white
          group=subtotal
```

```
            #1 @10 'Adams has' +1 a
            #2 @10 'Brown has' +1 b
            #3 @10 'Johnson has' +1 j
            #4 @10 'Smith has' +1 s
            #5 @10 'Other has' +1 o
            group=lastmessage
            #8 @10
            'ALL ARTICLES ASSIGNED.
            Press ENTER to stop processing.';
   display Assignment blank;
   if name='Adams' then a+1;
   else if name='Brown' then b+1;
   else if name='Johnson' then j+1;
   else if name='Smith' then s+1;
   else o+1;
   display Showtotal.subtotal blank noinput;
   if final then display Showtotal.lastmessage;
run;
```

When you execute the DATA step, the following windows appear.

In the Assignment window (located at the top of the display), you see the name of the article and a field into which you enter a reporter's name. After you type a name and press ENTER, SAS displays the Showtotal window (located at the bottom of the display) which shows the number of articles that are assigned to each reporter (including the assignment that you just made). As you continue to make assignments, the values in the Showtotal window are updated. During the last iteration of the DATA step, SAS displays the message that all articles are assigned, and instructs you to press ENTER to stop processing.

Example 3: Persisting and Nonpersisting Fields This example demonstrates the PERSIST= option. You move from one window to the other by positioning the cursor in the current window and pressing ENTER.

```
data _null_;
   array row{3} r1-r3;
   array col{3} c1-c3;
   input row{*} col{*};
   window One
            rows=20 columns=36
            #1 @14 'PERSIST=YES' color=black
            #(row{i}) @(col{i}) 'Hello'
            color=black persist=yes;

   window Two
            icolumn=43 rows=20 columns=36
            #1 @14 'PERSIST=NO' color=black
            #(row{i}) @(col{i}) 'Hello'
            color=black persist=no;
   do i=1 to 3;
      display One;
      display Two;
   end;
   datalines;
5 10 15 5 10 15
```

```
;
```

The following windows show the results of this DATA step after its third iteration.

Note that window One shows **Hello** in all three positions in which it was displayed. Window Two shows only the third and final position in which **Hello** was displayed.

Example 4: Sending a Message This example uses the _CMD_ and _MSG_ automatic variables to send a message when you execute an erroneous windowing command in a window that is defined with the WINDOW statement:

```
if _cmd_ ne ' ' then
   _msg_='CAUTION: UNRECOGNIZED COMMAND' || _cmd_;
```

When you enter a command that contains an error, SAS sets the value of _CMD_ to the text of the erroneous command. Because the value of _CMD_ is no longer blank, the IF statement is true. The THEN statement assigns to _MSG_ the value that is created by concatenating CAUTION: UNRECOGNIZED COMMAND and the value of _CMD_ (up to a total of 80 characters). The next time a DISPLAY statement displays that window, the message line of the window displays

```
CAUTION: UNRECOGNIZED COMMAND command
```

Command is the erroneous windowing command.

See Also

Statements:

"DISPLAY" on page 785

"The PMENU Procedure" in *SAS Procedures Guide*

X

Issues an operating-environment command from within a SAS session

Valid: anywhere
Category: Operating Environment

Syntax

X <'*operating-environment-command*'>;

Without Arguments

Using X without arguments places you in your operating environment, where you can issue commands that are specific to your environment.

Arguments

'*operating-environment-command*'
specifies an operating environment command that is enclosed in quotation marks.

Details

In all operating environments, you can use the X statement when you run SAS in windowing or interactive line mode. In some operating environments, you can use the X statement when you run SAS in batch or noninteractive mode.

Operating Environment Information: The X statement is dependent on your operating environment. See the SAS documentation for your operating environment to determine whether it is a valid statement on your system. Keep in mind:

- The way you return from operating environment mode to the SAS session is dependent on your operating environment.
- The commands that you use with the X statement are specific to your operating environment.

△

You can use the X statement with SAS macros to write a SAS program that can run in multiple operating environments. See SAS Guide to Macro Processing for information.

Comparisons

In a windowing session, the X command works exactly like the X statement except that you issue the command from a command line. You submit the X statement from the Program Editor window.

The X statement is similar to the SYSTEM function, the X command, and the CALL SYSTEM routine. In most cases, the X statement, X command or %SYSEXEC macro statement are preferable because they require less overhead. However, the SYSTEM function can be executed conditionally. The X statement is a global statement and executes as a DATA step is being compiled.

See Also

CALL Routine:

"CALL SYSTEM" on page 271

Function:

"SYSTEM" on page 561

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

SAS® Language Reference, Version 8

ISBN 1–58025–369–5

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

1st printing, October 1999